conference

proceedings

USENIX 1998 Annual Technical Conference

New Orleans, Louisiana
June 15-19, 1998

Sponsored by
The USENIX Association

The Advanced Computing
Systems Association

USENIX 1998 Annual Technical Conference

New Orleans, Louisiana June 1998

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: *office@usenix.org*
WWW URL: *http://www.usenix.org*

The price is $32 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1997 Anaheim
1996 San Diego
1995 New Orleans
1994 Summer Boston
1994 Winter San Francisco
1993 Summer Cincinnati
1993 Winter San Diego
1992 Summer San Antonio
1992 Winter San Francisco
1991 Summer Nashville
1991 Winter Dallas
1990 Summer Anaheim
1990 Winter Washington, D.C.
1989 Summer Baltimore
1989 Winter San Diego
1988 Summer San Francisco
1988 Winter Dallas
1987 Summer Phoenix
1987 Winter Washington, D.C.
1986 Summer Atlanta
1986 Winter Denver
1985 Summer Portland
1985 Winter Dallas
1984 Summer Salt Lake City
1984 Winter Washington, D.C.
1983 Summer Toronto
1983 Winter San Diego

ISBN 1-880446-94-4

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

The USENIX Association

Proceedings of the
USENIX 1998 Annual Technical Conference

June 15-19, 1998
New Orleans, Louisiana, USA

ACKNOWLEDGMENTS

Program Chair
Fred Douglis, *AT&T Labs – Research*

Program Committee
Pei Cao, *University of Wisconsin*
Ed Felten, *Princeton University*
Frans Kaashoek, *MIT*
Orran Krieger, *IBM*
Greg Minshall, *Fiberlane Communications*
Mike Nelson, *Silicon Graphics, Inc.*
Dave Presotto, *Lucent Technologies, Bell Labs*
Mike Schwartz, *@Home Network*
Terri Watson Rashid, *Microsoft Corporation*
Ben Zorn, *University of Colorado*
Elizabeth Zwicky, *Silicon Graphics, Inc.*

Invited Talks Coordinators
Clem Cole, *Digital Equipment Corporation*
Berry Kercheval, *Competitive Automation*

FREENIX Track Program Chair
Jon "maddog" Hall, *Digital Equipment Corporation*

FREENIX Track Program Committee
Theo De Raadt, *The OpenBSD Project*
Chris Farris, *Atlanta Linux Enthusiasts*
Greg Hankins, *Georgia Institute of Technology*
Jordan Hubbard, *The FreeBSD Project*
Poul-Henning Kamp, *The FreeBSD Project*
Angelos Keromytis, *University of Pennsylvania*
Marshall Kirk McKusick, *Author and Consultant*
Perry E. Metzger, *Piermont Information Systems*
Dan Shearer, *LAN Magazine*
Jason R. Thorpe, *The NetBSD Foundation, Inc.*
Theodore Ts'o, *MIT*

External Reviewers

Gaurav Banga
Brad Calder
Dan Duchamp
Dawson Engler
Trevor Fiatal
Benjamin Gamsa
Pawan Goyal
John Heidemann
Andrew Heybey
Andrew Hume
John Jannotti
Brian Kernighan
Eddie Kohler
Ram Kordale
P. Krishnan
Geoffrey H. Kuenning
James Larus
Han Lee
David Mazieres
Mark Mergen
Ethan Miller
Adam S. Moskowitz
Gerald Neufeld
Michael Rabinovich
K. K. Ramakrishnan
Rick Rashid
Matthew Seidl
Andrea Skarra
Mark Sullivan
Kiem-Phong Vo
Bruce Zenel

Works in Progress Coordinator
Terri Watson Rashid, *Microsoft Corporation*

USENIX Board Liaison
Andrew Hume, *AT&T Labs – Research*

Terminal Room
Lynda McGinley, *University of Colorado*

Administration: USENIX Association Staff
Ellie Young, *Executive Director*
Judy DesHarnais, *Conference Coordinator*
Daniel V. Klein, *Tutorial Director*
Cynthia Deno, *Marketing and Exhibitions*
Linda Barnett, *Marketing Communications*

USENIX Proceedings Production
Eileen Cohen, *Publications Director*
Data Reproductions

USENIX Support Staff
Eileen Curtis
Diane DeMartini
Jackson Dodd
Lana Erlanson
Julie Keiser
Toni Veglia

CONTENTS

Wednesday, June 17

Performance I

Session Chair: Fred Douglis, AT&T Labs – Research

Extensibility

Session Chair: Terri Watson Rashid, Microsoft Corporation

Commercial Applications

Session Chair: Dave Presotto, Bell Laboratories, Lucent Technologies

Thursday, June 18

Performance II

Session Chair: Mike Nelson, Silicon Graphics, Inc.

Neat Stuff

Session Chair: Pei Cao, University of Wisconsin

Friday, June 19

Networking

Session Chair: Elizabeth Zwicky, Silicon Graphics, Inc.

Real Time

Session Chair: Greg Minshall, Fiberlane Communications

Security

Session Chair: Fred Douglis, AT&T Laboratories

PREFACE

Welcome to the 1998 USENIX Technical Conference. I hope that for those of you not from the area, your visit to New Orleans is memorable for its cultural aspects as well as the technical ones that all attendees will encounter. I'm certainly looking forward to my own first visit to the region.

We have selected 23 technical papers for the refereed track this year, in addition to 28 separate papers in the FREENIX track and 13 invited talks. These are complemented by two days of tutorials, numerous BOFs, and a session of work-in-progress talks. Topics of refereed papers include performance, networking, security, extensibility, kernel issues, and other applications. This year we have three joint sessions with invited talks, ranging from The Amazing Randi's keynote on "Science and the Chimera," to "Historical UNIX," to "Wearable Computing."

Our program committee selected 23 of 87 submissions, using a total of 454 reviews (an average of over five reviews per paper) from the 12-person program committee and 31 external reviewers, collected in a five-week period. (During this time, PC members read about 30 papers each.) The original submissions came from 224 authors throughout the world, to whom we owe our gratitude for their contributions.

I would like to thank those who provided their assistance to me and the conference. There are no doubt others, not included here, who also deserve our thanks; I apologize in advance for any omissions. First of all, the USENIX staff has made organizing this conference a piece of cake: Ellie Young, Eileen Cohen, Cynthia Deno, Judy DesHarnais, Jackson Dodd, Zanna Knight, Linda Barnett, and Toni Veglia. Andrew Hume, the USENIX board liaison to the conference, provided excellent advice on selecting a program committee and other aspects of conference organization. Clem Cole and Berry Kercheval pulled together a great selection of invited talks. Jon "maddog" Hall chaired a separate program committee for FREENIX. Thanks to several past chairs for prompt and helpful advice: John Kohl, Jeff Mogul, and Margo Seltzer. Thanks to Rob Shillner for taking notes during our program committee meeting, and thanks to AT&T for hosting it; for providing administrative assistance in the form of Omni Bashorun and Tialynn Phillips, without whom the meeting could not have taken place; and for supporting my efforts as chair.

Most of all, thanks to the program committee and other reviewers for their diligent efforts. We received several compliments on the quality of the reviews, which I attribute to the many reviewers and their generous devotion of time and effort. Special credit goes to several external reviewers who reviewed six papers or more: John Heidemann, Geoffrey Kuenning, Ethan Miller, and Adam Moskovitz. The full lists of PC members and external reviewers appear on page ii.

Fred Douglis, Program Chair
April, 1998

AUTHOR INDEX

Scalable kernel performance for Internet servers under realistic loads

Gaurav Banga gaurav@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX, 77005

Jeffrey C. Mogul mogul@pa.dec.com
Digital Equipment Corp. Western Research Lab., 250 University Ave., Palo Alto, CA, 94301

Abstract

UNIX Internet servers with an event-driven architecture often perform poorly under real workloads, even if they perform well under laboratory benchmarking conditions. We investigated the poor performance of event-driven servers. We found that the delays typical in wide-area networks cause busy servers to manage a large number of simultaneous connections. We also observed that the *select* system call implementation in most UNIX kernels scales poorly with the number of connections being managed by a process. The UNIX algorithm for allocating file descriptors also scales poorly. These algorithmic problems lead directly to the poor performance of event-driven servers.

We implemented scalable versions of the select system call and the descriptor allocation algorithm. This led to an improvement of up to 58% in Web proxy and Web server throughput, and dramatically improved the scalability of the system.

1 Introduction

Many Web servers and proxies are implemented as as single-threaded event-driven processes. This approach is motivated by the belief that an event-driven architecture has some advantages over a thread-per-connection architecture [17], and that it is more efficient than process-per-connection designs, including "pre-forked" process-per-connection systems. In particular, event-driven servers have lower context-switching and synchronization overhead, especially in the context of single-processor machines.

Unfortunately, event-driven servers have been observed to perform poorly under real conditions. In a recent study of Digital's Palo Alto Web proxies, Maltzahn et. al. [11] found that the Squid (formerly Harvest) proxy server[5, 22] performs no better than the older CERN proxy[10]. This is surprising, because the CERN proxy forks a new process to handle each new connection, and process creation is a moderately expensive operation. This result is also in sharp contrast with the study by Chankhunthod et al.[5], which concluded that Harvest is an order of magnitude faster than the CERN proxy.

Maltzahn et. al. [11] attribute Squid's poor performance to the amount of CPU time Squid uses to implement its own memory management and non-blocking network I/O abstractions. We investigated this phenomenon in more detail, and found out that the large delays typical of wide-area networks (WANs) cause Squid to have a large number of simultaneously open connections. Unfortunately, the traditional UNIX implementations of several kernel features used by event-driven single-process servers do not scale well with the number of active descriptors in a process. These are the *select* system call, used to support non-blocking I/O, and the kernel routine that allocates a new file descriptor. (We refer to the descriptor-allocation routine as ufalloc(), as it is named in Digital UNIX, although other UNIX variants use different names, e.g., fdalloc().) A system running the Squid server spends a large fraction of its time in these kernel routines, which is directly responsible for Squid's poor performance under real workloads.

We designed and implemented scalable versions of select() and ufalloc() in Digital UNIX, and evaluated the performance of Squid and an event-driven Web server in a simulated WAN environment. We observed throughput improvements of up to 43% for the Web server, and up to 58% for Squid. We observed dramatic reductions in CPU utilizations at lower loads. We also evaluated these changes on a busy HTTP proxy server, which handles several million requests per day.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the working of a typical event-driven server running on a UNIX system. We also describe the dynamics of typical implementations of select() and ufalloc(). Section 3 describes our quantitative characterization of the performance problems in select() and ufalloc(). In Section 4 we present scalable versions of select() and ufalloc(). In Sections 5 and 6 we evaluate our implementation. Finally, Section 7 covers related work and and offers some conclusions.

2 Background

In this section we present a brief overview of the working of a typical event-driven server. We will also describe classical implementations of select() and ufalloc(). This will provide necessary background for the discussion in the following sections.

2.1 Event-driven servers

An event-driven server typically has a single thread which manages all connections to the server. The thread uses the select() system call to simultaneously wait for events on these connections.

When a call to select() returns, the server's main loop invokes event handlers for each of the ready descriptors. These handlers perform a variety of tasks depending on the nature of the particular event. For example, when a socket being used to listen for new connections becomes ready, the corresponding handler calls accept() to return a file descriptor for the new connection. Handlers invoked when a connection becomes ready for reading or writing perform the actual read or write to the appropriate descriptor. The execution of handlers may cause the addition or removal of descriptors from the set being managed by the server.

Event-driven servers are fast because they have no locking or context switching overhead. The same thread manages all connections, and all handlers are executed synchronously. A single-threaded server, however, cannot exploit any true concurrency in the stream of tasks. Thus, on multiprocessor systems, event-driven servers have as many threads as processors. Examples of event-driven servers include Squid[5, 22] and its commercial version NetCache[16], Zeus[25], thttpd[24] and several research servers[2, 8, 18].

2.2 select()

The *select* system call allows a user process to wait for events on a set of descriptors. A process can indicate interest in three types of events on a descriptor: events that make a descriptor *readable*, those that make it *writable*, and *exception* events. This information is passed to the kernel using three bitmaps. In each bitmap the kth bit indicates interest in events of that type for the kth descriptor. These bitmaps are value-result parameters, and the returned bitmaps indicate the sets of ready descriptors. Stevens[23] describes the select() interface in detail.

We describe the Digital UNIX implementation of select(). However, the classical BSD implementation of select() is similar to the Digital UNIX implementation. The main differences are related to the multithreaded nature of the Digital UNIX kernel. Thus our discussion is fully applicable to 4.3BSD and most BSD-derived implementations. Also, we discuss how select() works for descriptors that represent sockets, but our discussion and algorithms can be trivially extended to include descriptors that refer to other kinds of objects, such as vnodes. (Vnodes are kernel data structures used to represent files and devices.)

In Digital UNIX, the select() function in the kernel starts by creating internal data structures containing summary information about sockets that are marked in at least one input bitmap. Subsequently, select() calls do_scan(), which calls selscan() to check the status of each of the entities (vnodes or sockets) corresponding to the selected descriptors.

For each selected socket, selscan() enqueues a record referring to the current thread on the *select queue* of the socket. This is done so that the thread can be identified as waiting inside select() for events on the socket. selscan() then calls soo_select() for each socket, which checks to see if the condition that the process is interested in (i.e. the socket is readable, writable, or has pending exceptions) is true. If none of the conditions that the user process is selecting on are true, then do_scan() goes to sleep waiting for any of these to become true.

Note that the linear search in selscan() covers every socket of potential interest to the selecting process, independent of how many are actually ready. Thus, the cost is proportional to the number of file descriptors involved in the call to select(), rather than to the number of events discovered by the call.

When a network packet comes in, protocol processing may cause a condition on which do_scan() is blocked to become true. The thread that performs protocol processing for an incoming packet calls select_wakeup(), which wakes up all threads that are blocked in do_scan() awaiting this condition.

A thread that is woken up in do_scan() calls selscan(), which calls soo_select() for *all* the sockets that the corresponding call to select() specified in its three bitmaps. do_scan() also calls undo_scan() to remove this thread from select queues of the selected sockets.

2.3 ufalloc()

The kernel function ufalloc() is called to allocate a new file descriptor for a process. This function is called as a result of the open(), socket(), socketpair(), dup(), dup2() and accept() system calls.

UNIX semantics for file descriptor allocation require that the kernel allocate the lowest-numbered available descriptor. This prevents the use of a straightforward scalable implementation, such as a free list. Instead, all of the UNIX variants that we know of, including BSD-derived systems such as Digital UNIX, and System V Release 4 systems such as Solaris, use a linear search of the file descriptor table. The search starts with file descriptor 0 and continues to the first NULL entry. The cost of this search is roughly proportional to the number of open file

descriptors, although it might complete before checking all of the possible descriptor table slots.

3 Problems in select() and ufalloc()

As we observed in section 1, Maltzahn et. al. [11] found that the Squid proxy server performs no better than the older CERN proxy under real workloads, contradicting the study by Chankhunthod et al.[5], which concluded that Harvest is an order of magnitude faster than the CERN proxy. Indeed, a simple LAN-based experiment using a simulated client load does show a big performance difference between Squid and the CERN proxy.

In an attempt to explain this peculiar result, we tried to understand why Squid's performance under real load is so much worse than under ideal conditions. One factor that is different in the two scenarios is that under real load Squid manages a much larger number of simultaneous connections than in a LAN-based test scenario. This is because of much larger delays experienced in WANs. Because WAN environments have larger round-trip times (RTTs), and are more likely to exhibit packet losses, HTTP connections tend to last much longer in WAN environments than in simple LAN environments. Therefore, for a given connection arrival rate, a WAN-based HTTP server will have more open connections than a server in a LAN environment.

Richardson's measurements of Digital's Palo Alto Web proxies [19] show between 30 and 950 simultaneously open connections, depending on time of day. Richardson's measurements also show that while the median response time is about 250 msec., the mean is 2.5 seconds: some connections stay open for a very long time. The large ratio of mean to median holds over a wide range of response sizes (although the 10:1 ratio only holds when all response sizes are considered together). This implies that at any given time, most of the open connections are *cold* (idle for long intervals), and only a few are *hot*.

Following this intuition, we tried to evaluate the effect of a large number of cold connections on Squid performance. We used DCPI [1] to profile a system running the Squid proxy under a carefully designed request load. To simulate the effect of large WAN delays, we set up a dummy HTTP client process on a client machine. This process opened a large number (100-2000) of connections to the Squid server but subsequently made no requests on these connections. We refer to this process as the *load-adding client*. Another process on the client machine simulated a small number (10-50) of HTTP clients, which repeatedly made HTTP requests of the proxy. Each request retrieved a 1259-byte response. We used the scalable client (S-Client) architecture from Banga and Druschel [3].

In our tests, we ran the Squid server process on an AlphaStation 500 (400Mhz 21164, 8KB I-cache, 8KB D-cache, 96KB level 2 unified cache, 2MB level 3 unified cache, SPECint95 = 12.3) equipped with 192MB of physical memory. The server operating system was Digital UNIX 4.0B, with the latest patches that were available at the time. The client machine was a 333Mhz AlphaStation 500 (same cache configuration as above, SPECint95 9.82) with 640MB of physical memory, running DUNIX 3.2C. The Squid version used was Squid-1.1.11. The client and server were connected using a 100Mbps FDDI network.

This experiment indicates that up to 53% of the system's CPU time is being spent inside select() (and its various components – selscan(), soo_select(), etc.). Up to 11% of the CPU is being spent by the user process in collating information from the bitmaps returned by select().

Our detailed results are shown in Figure 1. The x-axis represents the number of cold connections. Curves are plotted, for both 10 hot connections and 50 hot connections, showing the percentage of CPU time spent in kernel-mode functions related to select(), and the percentage of CPU time spent in the user-mode select() loop.

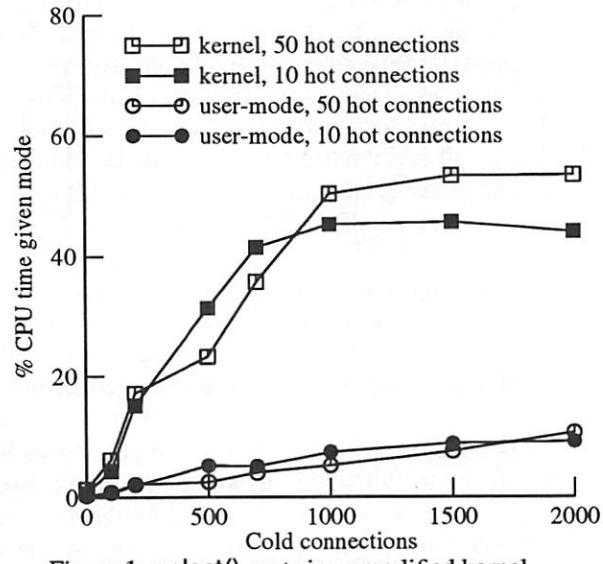


Figure 1: select() costs in unmodified kernel

Figure 1 shows that the costs of both the kernel select() implementation and the user-mode select() loop rise significantly with increasing numbers of cold connections. Also, these costs are relatively independent of the number of hot connections, up to about 1000 cold connections.

The costs are initially linear in the number of cold connections, but eventually they flatten out. As the number of cold connections increases, the system spends more

CPU time in each call to select(), and so the calls to select() come less often. This causes the number of pending events returned by select() to increase (at low loads, select() usually returns just one pending event, but when called infrequently, it often returns several). The cost of each select() call is thus amortized over a larger number of interesting events. Thus, the total CPU cost of select(), which is proportional to the number of select()s per second times the cost of each select, tends to level off.

These numbers were generated with a request load of about 100 requests/second. At higher rates, select() is still important, but ufalloc() also consumes significant CPU time, because of its linear search algorithm. A typical DCPI profile for the system above, with 750 cold connections, 50 hot connections, and 220 new connections/second, is shown in Table 1.

CPU %	Procedure	Mode
21.91%	*all kernel* select *functions*	kernel
8.31%	soo_select()	kernel
7.56%	selscan()	kernel
4.82%	undo_scan()	kernel
1.22%	select()	kernel
17.79%	ufalloc()	kernel
4.23%	comm_select()	user
1.71%	_Xsyscall()	kernel
1.68%	_doprnt()	user
1.32%	idle_thread()	kernel
1.20%	memset()	user
1.15%	cache_lookup()	kernel
1.10%	namei()	kernel

750 cold connections, 50 hot connections, 220 requests/second

Table 1: Example profile for unmodified kernel

In summary, the current implementations of select() and ufalloc() do not scale well with the number of open connections in a server process. Both algorithms do work that is linear in the number of connections being managed by the process, and proxies in WAN environments tend to have many open connections. In the next section we will describe our implementation of scalable versions of these functions.

4 Scalable select() and ufalloc()

In this section we describe our design for scalable versions of select() and ufalloc(). We also describe our prototype implementation of these designs in Digital UNIX.

4.1 select()

Consider an event-driven server process waiting for activity on any of a few thousand sockets. Recall from Section 2 that select() always performs a full scan through all of these sockets, either to find those few that are currently ready, or to indicate that a thread is waiting for events on each of the sockets.

A full scan is also performed after the protocol code processes an incoming packet and calls select_wakeup() to unblock a thread waiting inside select(). The full scan is performed even though only a few of the sockets are actually ready. This wasted effort is expended because, between the call to select_wakeup() and the invocation of do_scan(), we throw away the information about the identity of the socket that has become ready. selscan() then does a significant amount of work to rediscover the set of ready sockets.

The key idea of our design is to preserve information about the change in the state of a socket between select_wakeup() and do_scan(). We use this information to prune both the initial scan, and the scan after the select_wakeup(), to inspect only those sockets that need inspection. These are the sockets either about which we have no prior information, or for which we have state-change hints from the protocol-processing layer.

We changed the Digital UNIX kernel to keep track of three sets for each thread, named READY, INTERESTED, and HINTS. (The first two of these sets actually consist of three component sets, one for read-ready descriptors, one for write-ready descriptors and one for exceptions.) The INTERESTED set is the subset of sockets that the thread is currently interested in selecting on. The READY set is a subset of the INTERESTED set and includes those sockets which the kernel thinks are *ready*. The kernel maintains state-change information about sockets in the INTERESTED set, rather than for the full set of sockets open for a thread. This state-change information is maintained as the HINTS set. The HINTS set includes sockets that might have become ready since the last call to select(), and is updated by the protocol layer when a packet arrives for a socket.

Each call to select() specifies a SELECTING set for the thread, which is used to compute the new values of the READY and INTERESTED sets. select() uses the HINTS and READY sets to prune its initial scan. It checks only those sockets which are in the SELECTING set and either:

1. are not in the old INTERESTED set, or
2. are in the old READY set, or
3. are in the HINTS set

Mathematically, we can express the computation of

these sets as:

$$INTERESTED_{new} = SELECTING \cup INTERESTED_{old}$$

$$READY_{new} = \mathcal{C}(INTERESTED_{new} \cap (\overline{INTERESTED_{old}} \cup READY_{old} \cup HINTS))$$

where $\mathcal{C}$ expresses the computation of checking the status of descriptors in its argument set.

The computation of $\mathcal{C}$'s argument set above appears to have complexity proportional to the size of the SELECTING set. We took care to optimize this computation and its data-cache footprint. The resulting code has a very small cost relative to other parts of select().

The set returned from select() is:

$$READY_{to\_user} = SELECTING \cap READY_{new}$$

A descriptor must be removed from the INTERESTED sets of *all* threads in a process at some point between the time that the descriptor is closed and the time that it is next allocated by *any* thread in the process.

For each socket, we record the set of processes that have a reference to the socket. In the protocol processing code, when a packet comes in for a socket, sowakeup() records a hint in the HINTS sets of each of the threads in the referencing processes for which this socket is present in the INTERESTED set of the thread. sowakeup() also wakes up all such threads that are blocked in select(). After a thread is woken up in select(), it scans only those sockets in its HINTS set.

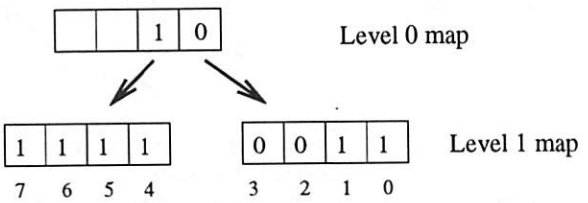


Figure 2: Two-level ufalloc bitmap

4.2 ufalloc()

The existing ufalloc() implementation uses a linear search to find the lowest-numbered free descriptor. We converted this into a logarithmic-time algorithm by adding an auxiliary data structure, a two-level tree of bitmaps. The collection of all the level-1 nodes can be thought of as a single bitmap; each bit in this bitmap describes the allocation state of one file descriptor. One-valued bits in this bitmap correspond to allocated descriptors. The level-1 bitmap is stored as an array of nodes.

Each bit in the level-0 bitmap describes the state of an entire level-1 node. One-valued bits in this bitmap correspond to level-1 nodes with no zero bits; a zero-valued bit in the level-0 bitmap corresponds to a level-1 node with at least one zero bit.

Figure 2 shows an example of such a tree. For simplicity, this figure depicts the nodes as 4-bit integers, although our actual implementation uses 64-bit integers. We use the Alpha's little-endian bit-order in this example. The example tree shows that descriptors 0, 1, and 4 through 7 are allocated, while descriptors 2 and 3 are free.

When a process wants to allocate a new file descriptor, the level-0 bitmap is searched for the first zero bit. The index of this bit is used as an index into the array of level-1 nodes, and the indexed node is then searched to find the first zero bit. Efficient algorithms exist for finding the first zero bit in a word, but we have found that a simple linear search is sufficiently fast, since the dominant cost on modern CPUs is the number of data-cache misses, not the number of instructions executed.

When a descriptor is deallocated, the appropriate bits are cleared in both bitmaps. This leads to a constant-time cost for deallocation.

With the level-1 nodes and the entire level-0 bitmap represented as 64-bit words, this algorithm directly supports 4096 descriptors per process. A straightforward generalization to a deeper tree would support an enormous number of descriptors, even if a smaller word size were used.

5 Experimental Evaluation

We evaluated the effects of our implementation of select() and ufalloc() on the performance of two event-driven Internet servers: the Squid proxy, and the thttpd [24] Web server (we used a modified version of thttpd with numerous performance improvements [18]). These experiments were performed using the same server and client systems describe in Section 3. We also measured the effect of our changes on the performance of Digital's Palo Alto proxies.

5.1 Scalability with respect to connection rate

The S-Client architecture introduced by Banga and Druschel [3] allows the generation of high HTTP request rates, using a small number of client machines. We used S-Clients to vary the load on the server. At the lowest load, the server is underutilized; at the higher loads, the server is the bottleneck.

For each request rate, we ran two kinds of benchmarks. In the naive benchmark, we used only enough S-

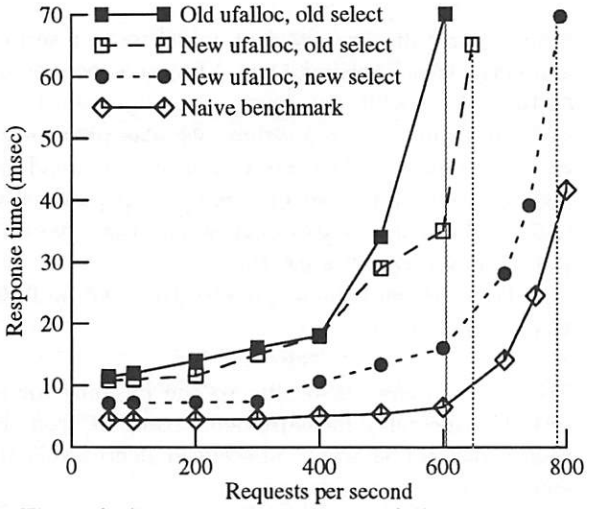


Figure 3: Squid response times – 1259-byte files

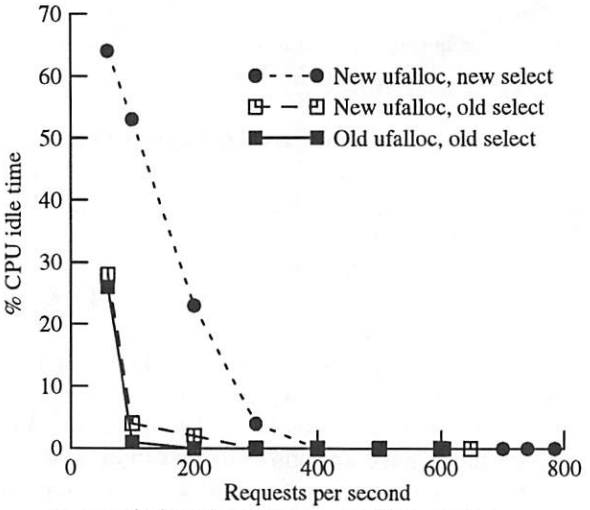


Figure 4: Squid idle time – 1259-byte files

Clients to generate the desired request rate. In the more realistic benchmark, we also used a load-adding client, to simulate the presence of long-delay connections. The load-adding client was run with 750 infinitely slow connections. (We show the effect of varying the number of slow connections in Section 5.2.)

All clients, in all of the experiments, repeatedly requested a single file of a fixed sized. In some experiments, we used an 8192-byte file; this is within the range of typical response sizes reported for the Web. In other experiments, we used a 1259-byte file; the shorter file size places more emphasis on per-connection overheads.

For our experiments using the Squid proxy server, we arranged things so that each request received by the proxy would generate an "If-Modified-Since" message from the proxy to the origin server, but the actual data would be served from the proxy's cache. The origin server ran on identical hardware (a 400Mhz AlphaStation 500), using the thttpd server program; we ensured that the origin server was never the bottleneck.

Figure 3 shows how the response time of the Squid proxy varies with request rate, for 1259-byte files. The results for all kernels on the naive benchmark are effectively identical; for the realistic benchmark, we plot different curves for the different kernels. For each curve, the final point shows the "saturation throughput" for the given kernel; beyond this point, increasing the offered load did not increase throughput. This figure clearly shows that the presence of adding slow connections in the realistic benchmark drastically reduces the throughput achieved with the unmodified kernel relative to the naive benchmark. It also shows that our new implementations of select() and ufalloc() solve this performance problem. The performance of the fully modified kernel is nearly independent of the presence of many slow connections.

Figure 4 shows the effect of the new versions of select() and ufalloc() on server CPU idle time, also for 1259-byte files. At lower request rates, where the server is underutilized, our modifications greatly increase idle time for the realistic benchmark. The increase in idle time reflects the improved scalability of the system in the presence of cold connections.

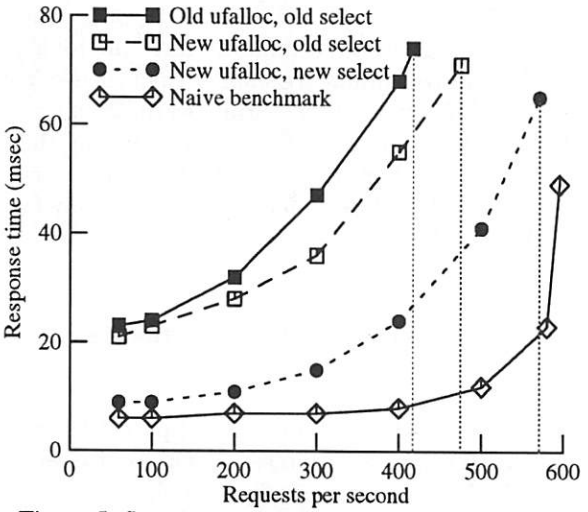


Figure 5: Squid response times – 8192-byte files

Figure 5 shows the response time of the Squid proxy for 8129-byte files. As in Figure 3, the fully modified kernel provides a higher saturation request rate than the original kernel, and yields lower response times at all request rates. However, the new kernel's performance on the realistic benchmark does not come quite as close to the performance of the naive benchmark; this may be due to data-cache collisions between the larger packets and the kernel's data structures. In these tests, the unmodified kernel showed no idle time for all request rates, while the new kernel showed some idle time up to 300

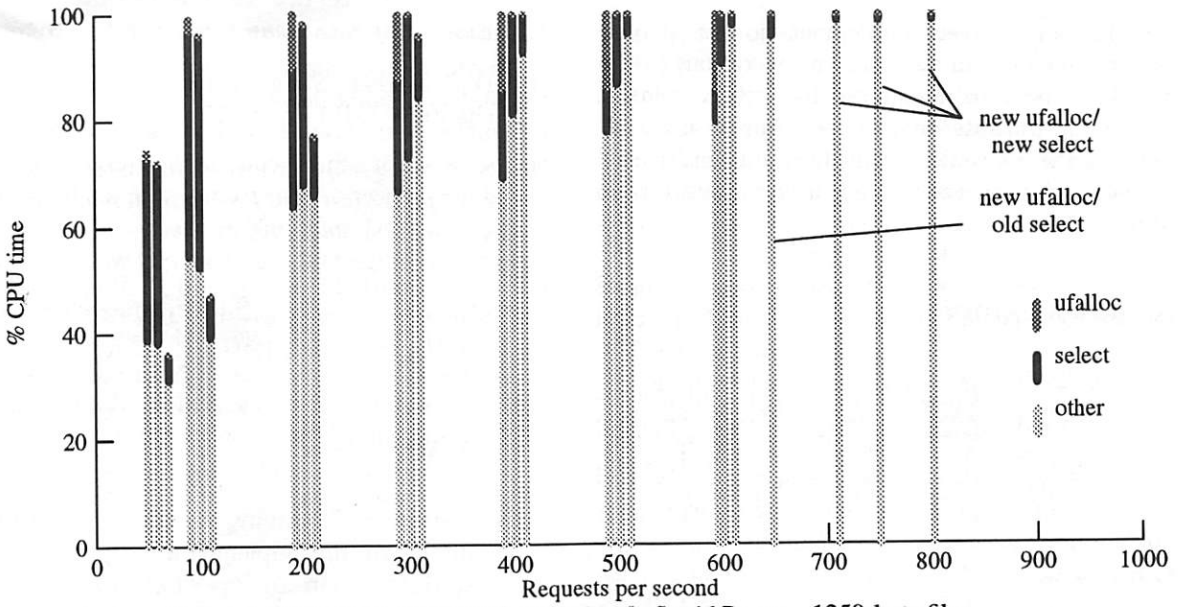


Figure 6: CPU share of ufalloc() and select(), Squid Proxy – 1259-byte files

requests/sec.

We used DCPI to obtain CPU time profiles of the server. Figure 6 shows the fraction of CPU time used in select() and in ufalloc(), for various request rates, using 1259-byte files. (The results for tests using 8192-byte files are analogous.) In each group of three bars, the leftmost bar represents the unmodified kernel, the center bar represents the kernel with the new select(), and the rightmost bar represents the kernel with new versions of both select() and ufalloc(). At rates above 600 requests per second, each bar is independently labelled. The top section of each bar shows the CPU time spent in ufalloc(), and the middle section shows the CPU time spent in select(). The bottom section of each bar ("others") shows the CPU time used for all other components of the server, including user-mode code. Idle time is not shown; it corresponds to the space above the bar, if any.

Figure 6 shows that the new ufalloc() almost entirely eliminates the CPU costs of descriptor allocation in all of the tested configurations. The new select() also costs much less than the old select().

When the server is underutilized, at rates below about 200 requests per second, the CPU profiles show that the new select() provides an additional performance impact: although we have not changed the implementation of any code covered by the "others" part of the profile, and the total throughput has not changed, the CPU costs of the "others" components has been reduced, relative to the unmodified kernel. We attribute this to better data-cache behavior, because the new select() has a much smaller data-cache footprint than the original implementation. The modified ufalloc() may also have a similar effect on cache performance. The improved data-cache footprint of select() is probably responsible for some of the throughput gains in the server-bound configurations.

CPU %	Procedure	Mode
21.96%	*all idle time*	kernel
11.49%	*all kernel* select *functions*	kernel
11.24%	select()	kernel
0.15%	new_soo_select()	kernel
0.10%	new_selscan_one()	kernel
16.37%	comm_select()	user
2.61%	tcp_slowtimo()	kernel
1.73%	tcp_fasttimo()	kernel
1.39%	_doprnt()	user
1.21%	_Xsyscall()	kernel
1.10%	_XentInt()	kernel
1.00%	bcopy()	kernel
0.91%	read_io_port()	kernel
0.90%	memset()	user

750 cold connections, 50 hot connections, 220 requests/second

Table 2: Example profile for modified kernel

As can be seen in Figure 3, even with our kernel modifications, the realistic benchmark still causes a small performance degradation compared to the naive benchmark. We attribute this to the inherently poor scalability of the select() programming interface. This interface passes information proportional to the total number of active connections on each call to select(). Moreover, when

select() returns, the user process must do work proportional to the total number of active connections to discover which descriptors have pending events. Finally, select() overwrites its input bitmaps, thus requiring additional user-mode work to create these bitmaps on each call. These costs cannot be eliminated with the current interface. In a separate publication [4], we propose a new, scalable interface to replace select()

Table 2 shows a profile of the modified kernel, made under the same conditions as the profile of the original kernel shown in Table 1. The new kernel spends 22% of the time in the idle loop, compared to almost no idle time for the original kernel. The original kernel spent about 22% of the CPU in select() and its subroutines, and 18% of the CPU in ufalloc(). The modified kernel spends 11% of the CPU in select(), and virtually none in ufalloc(). However, the busiest function in the system is now the user-level comm_select() function, using 16% of the CPU. The almost 28% of the CPU together consumed by the kernel select() and user-mode comm_select() functions is a result of the poorly scaling bitmap-based select() programming interface.

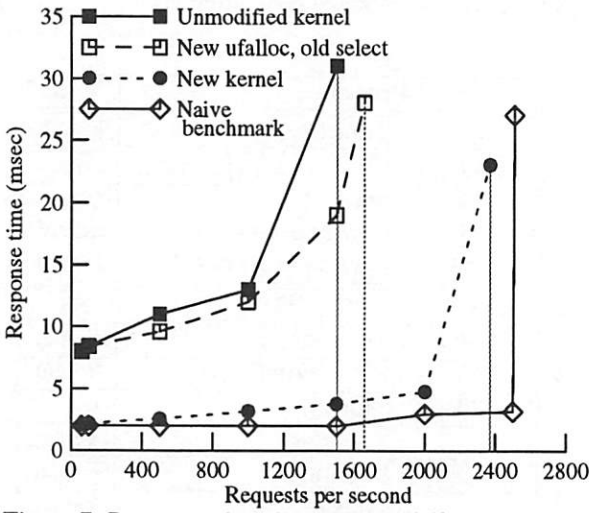


Figure 7: Response time for thttpd – 1259 byte files

Our experiments using the thttpd [24] Web server gave similar results. Using our modified kernel (with new implementations of both select() and ufalloc()), server throughput (at server saturation) improved by 58% for 1259-byte files, as shown in figure 7. For 8192-byte files, throughput increased by 37%; further improvement may have been limited by the available network bandwidth, rather than by the server. At lower request rates, the modified kernel showed much more idle time. For example, at 100 requests/sec. for a 1259-byte file, the unmodified kernel showed 16% idle time; the modified kernel showed 88% idle time. At at 100 requests/sec. for an 8192-byte file, the unmodified kernel had no idle time, but the modified kernel still showed 73% idle time.

5.2 Scalability with respect to connection count

To demonstrate that our implementations of select() and ufalloc(), unlike the original code, does scale well as the number of cold connections increases, we performed another series of experiments. In these experiments, we varied the number of connections from the load-adding client, between 0 and 2000 connections, and then increased the request rate until the server was saturated.

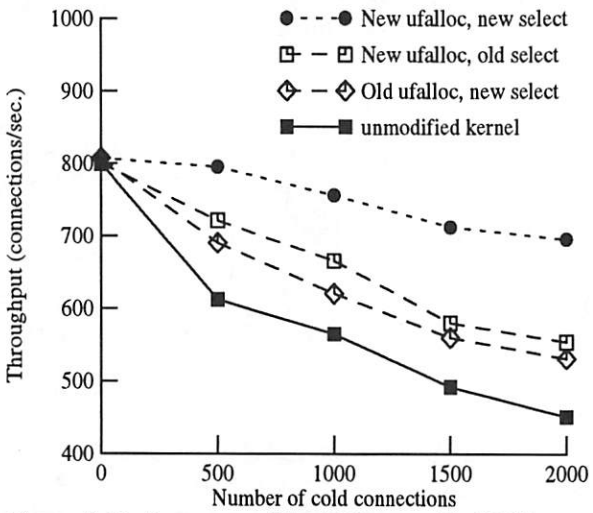


Figure 8: Performance of Squid Proxy – Scalability

Figure 8 shows that the throughput of the original kernel drops by 44% as the number of of cold connections increases from zero to 2000. The figure also shows that the kernel with our scalable ufalloc() has a somewhat smaller dependency on the number of cold connections, and for the kernel with our implementations of both select() and ufalloc(), its throughput drops by only 14% over the same range. We believe that the remaining dependency results from the user-level costs of the programming interface for select().

6 Performance of a live system

Digital Equipment Corporation operates a Web proxy system, in Palo Alto, California, that serves a large fraction of Digital's internal users. During a typical weekday, the system handles as many as 2.6 million HTTP requests, from at least 5570 individual client hosts.

We installed our modified kernel on the proxy server, a 500 MHz AlphaStation 500 system (21164A processor, SPECInt95 = 15.0) with 512 MBytes of RAM. We then ran the system using either the unmodified kernel or our modified kernel, each for an entire calendar day (midnight to midnight, Pacific Time), and collected extensive monitoring information.

During these these tests, the proxy server used version 3.1.2c-OSF of the NetCache software [16] from Network Appliance, Inc. Like Squid, NetCache was based on the Harvest Cache software, although NetCache and Squid

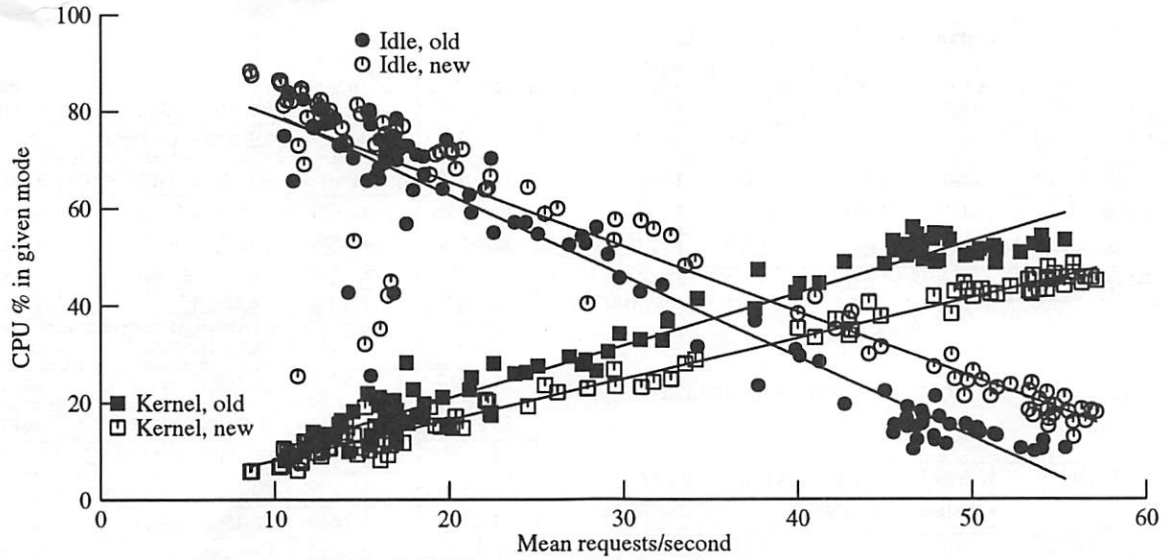


Figure 9: CPU costs as a function of request rate

Date	Kernel version	Requests handled	Max. alloc. fds	Peak req. rate
1998-04-16	old	2581113		107
1998-04-23	new	2602448	755	116

Table 3: Statistics for live tests

have since evolved separately. Because caching tends to reduce the number of simultaneous network connections, during our trials we operated this software with caching disabled. This increases the load on the system, but for various reasons does not significantly increase response time as seen by the users.

Table 3 shows some statistics for each of the trials. The "Max. alloc. fds" column shows the largest number of file descriptors allocated to a single process at any one point during the trial; the "Peak req. rate" column shows the largest number of requests logged during a single second over the course of the day.

6.1 Effect of request rate on CPU load

The operating system maintains counts of the number of clock interrupts that occur in each system mode (user-mode, kernel-mode, and idle). During the course of each trial, we logged these counters every 15 minutes, which allowed us to reconstruct the mean time spent in each mode during the 15 minutes prior to each log entry. The proxy software creates a timestamped log entry for each HTTP request it receives, so we can also count the number of requests handled in each 15 minute period, and then compute the mean request rate over that period.

Figure 9 shows how CPU idle time, and CPU kernel-mode time, vary as a function of the mean request rate. Each point on the scatterplot represents one 15-minute sample. The circles correspond to idle time; the squares correspond to kernel-mode time. The filled marks show performance with the old versions of both select() and ufalloc() (the trial of 1998-04-16). The open marks show the performance of the new implementations (the trial of 1998-04-23).

We then computed linear regressions for each set of samples. The regression lines are shown in Figure 9; the numeric results are given in Table 4. (User-mode regressions are given in the table, but not shown in the figure.) Each sample set includes 96 points (24 hours of 15-minute samples). The correlation between kernel-mode time and request rate is quite close; the correlation for idle time is not quite as good, probably because of some outliers caused by daily "housekeeping" tasks done during periods of low request rate. Because the outliers all occur at low request rates (that is, late at night), we recalculated the regressions after excluding samples taken at rates below 20 requests/second. These regressions, shown in Table 5, show higher correlation coefficients for idle time and user-mode time.

The regressions for idle time and kernel-mode time show significantly steeper slopes for the unmodified kernel, compared to those for the new implementations of select() and ufalloc(). The regressions for user-mode time suggest that the new kernel performs slightly better, perhaps because of better data-cache utilization, but the difference might not be significant.

Although one cannot necessarily expect linear behavior at very high request rates, a linear extrapolation of the idle time regressions from the full data sets gives X-intercepts of 58 requests/sec. for the unmodified kernel,

Date	Kernel version	CPU mode	Slope	Corr. coeff.
1998-04-16	old	idle	-1.67	-0.96
1998-04-23	new	idle	-1.34	-0.92
1998-04-16	old	kernel	1.09	0.98
1998-04-23	new	kernel	0.85	0.99
1998-04-16	old	user	0.58	0.77
1998-04-23	new	user	0.49	0.66

N = 96

Table 4: Linear regressions: full 1-day data sets

Date	Kernel version	CPU mode	Slope	Corr. coeff.
1998-04-16	old	idle	-1.69	-0.97
1998-04-23	new	idle	-1.46	-0.98
1998-04-16	old	kernel	1.02	0.96
1998-04-23	new	kernel	0.85	0.99
1998-04-16	old	user	0.68	0.97
1998-04-23	new	user	0.65	0.99

N = 54

Table 5: Linear regressions: above 20 requests/second

and 69 requests/sec. for the new implementation. Using the truncated data sets (Table 5), the calculated X-intercepts are 57 and 68 requests/sec., respectively. This suggests that the modified kernel might support a peak request rate about 19% higher than the unmodified kernel, in this application.

Note that our samples were averaged over 15-minute intervals. The actual one-second peak rates experienced during these trials (see Table 3) were 107 requests/sec. for the unmodified kernel, and 116 requests/sec. for the modified kernel. Clearly, the systems can support rates higher than the extrapolation of idle time implies. The main significance of our performance improvements may be not the increase in peak throughput, but the decrease in queueing delay (and response time) at high throughputs.

6.2 Profile results

We obtained CPU-time profiles, using DCPI, for the proxy server during periods of heavy load, for both the original kernel (Table 6) and our modified kernel (Table 7). Each profile covers a period of exactly one hour. The tables include all procedures accounting for at least 1% of the non-idle CPU time.

The first column in each profile shows the fraction of CPU time spent in each function or group of procedures.

CPU %	Non-idle CPU %	Procedure	Mode
10.77%		all idle time	kernel
89.23%	100.00%	all non-idle time	kernel
35.27%	39.53%	all select functions	kernel
13.51%	15.14%	selscan	kernel
12.56%	14.08%	soo_select	kernel
7.48%	8.38%	undo_scan	kernel
1.64%	1.83%	select	kernel
12.64%	14.17%	commSelect	user
1.74%	1.95%	all TCP functions	kernel
1.49%	1.67%	malloc-related #1	user
1.39%	1.56%	malloc-related #2	user
1.09%	1.22%	mutex_unblock	user
1.03%	1.16%	read_io_port	kernel
0.95%	1.07%	bcopy	kernel
0.94%	1.05%	memGrep	user

Profile on 1998-04-16 from 10:00 to 11:00 PDT
mean load = 54 requests/sec.
peak load ca. 98 requests/sec

Table 6: Profile of unmodified kernel on live proxy

As the first row in each table shows, even during periods of heavy load, some time is spent in the kernel's idle thread and its children. Therefore, the second column shows the fraction of non-idle CPU time spent in all non-idle procedures; this is a more useful basis for comparing the two kernels. Note that the profiles include a mixture of kernel-mode and user-mode procedures.

The modified kernel spends 30% of the non-idle CPU time in select() and related procedures, compared to almost 40% spent in such procedures by the unmodified kernel. However, kernel-mode select() processing is still a significant burden on the CPU. As in Figure 2, considerable time is spent in the user-mode commSelect() procedure (Squid and NetCache apparently use slightly different names for the same procedure). These observations support our belief that the bitmap-based select() programming interface leads to unnecessary work, and probably to significant capacity misses in the data caches.

In experiments with simulated loads, we observed that NetCache on our kernel calls select() about 7 times as it does on the unmodified kernel. We believe this is because our faster select() causes a NetCache thread to return from select() with usually only one ready descriptor[1]. Before the next event arrives, other NetCache threads call select() to discover this event again. In the unmodified kernel, each call to select() takes

[1] NetCache uses multiple event-driven threads, presumably for exploiting the parallelism available on SMP machines.

CPU %	Non-idle CPU %	Procedure	Mode
16.29%		all idle time	kernel
83.71%	100.00%	all non-idle time	kernel
25.11%	30.00%	all select functions	kernel
11.23%	13.42%	new_soo_select	kernel
7.73%	9.24%	new_selscan_one	kernel
5.67%	6.77%	select	kernel
0.04%	0.05%	new_undo_scan	kernel
15.33%	18.32%	commSelect	user
2.70%	3.23%	all TCP functions	kernel
2.56%	3.05%	in_pcblookup	kernel
1.09%	1.30%	mutex_unblock	user
1.01%	1.21%	bcopy	kernel
1.00%	1.19%	read_io_port	kernel
0.97%	1.16%	malloc-related #1	user
0.93%	1.12%	memGrep	user
0.91%	1.09%	malloc-related #2	user

Profile on 1998-04-23 from 10:00 to 11:00 PDT
mean load = 55 requests/sec.
peak load ca. 116 requests/sec

Table 7: Profile of modified kernel on live proxy

longer, and returns multiple events. This may account for the heavy use of select() in Table 7.

In this application, even the unmodified kernel spends very little time in ufalloc() (0.20%). However, the modified kernel spends even less time in ufalloc() (0.03%). For this proxy, the total number of open file descriptors is relatively small. However, one might expect this fraction to become more significant at higher request rates.

We are not entirely sure what caused the significant increase in time that the modified kernel spends in in_pcblookup. This may be the result of an unfortunate collision in the direct-mapped data caches.

We note that in this real-world environment, for both versions of the kernel, just over 1% of the non-idle CPU time is spent in all kernel-related data movement (the bcopy()). Even less time is spent computing checksums. A moderate amount of time (between 2% and 3%) is spent in TCP-related functions (which have been highly optimized in Digital UNIX). These measurements reinforce the emphasis placed by Kay and Pasquale[9] on "non-data touching processing overheads"; however, they failed to recognize that the poor scalability of select() would ultimately dominate the other costs.

6.3 Data cache effects

We have speculated in several places that our kernel modifications affect data cache utilization. DCPI allows us to estimate the mean cycles per instruction (CPI) for each procedure in a profile, and to estimate the fraction of dynamic stalls caused by data-cache misses. We found that the CPI for the user-mode commSelect() procedure declined from 1.69 to 1.62 as a result of our kernel changes, mostly because of fewer data-cache misses.

We also found that the CPI for in_pcblookup() increased from about 1.28 to 11.15 as an apparent result of our kernel changes, even though we did not change the code for this kernel procedure. This suggests that we somehow created a particularly unlucky collision in the data caches between the data structures for in_pcblookup() and those for select().

7 Related Work

Operating system researchers and vendors have devoted much effort to improving Internet server performance. One early experience that lead to published results was the 1994 California election server [14, 15]; another early study was performed at NCSA [12]. Operating system vendors responded to complaints of performance problems by improving various kernel mechanisms, especially by replacing BSD's linear-time PCB lookup algorithm [13, 21], and by changing certain kernel parameter values. Vendors also provided tuning guides for systems being used as Web servers [6].

In response to observations about the large context-switching overhead of process-per-connection servers, recent servers [5, 16, 22, 24, 25] have used event-driven architectures. Measurements of these servers under laboratory conditions indicate an order of magnitude performance improvement [5, 20].

Maltzahn et. al. [11] reported the poor performance of Squid under real conditions. Fox et al. [7], in describing the Inktomi system, also briefly mention that their event-driven front-ends spend 70% of their time in the kernel, and attribute this to the state-management overhead of a large number of simultaneous connections. However, neither of these papers analyzed the reason for this phenomenon in any detail.

8 Conclusion

We presented a detailed analysis of the effect of WAN delays on the performance of event-driven servers, and showed that linear scaling in the select() and ufalloc() implementations leads to excessive kernel CPU consumption.

We described scalable versions of select() and ufalloc(), and evaluated their impact on the performance of event-driven servers. We showed that these changes improve the performance of Web servers and proxies on realistic benchmarks, and on a live proxy, without harming performance on naive benchmarks.

Our results show the need for a new, scalable interface

to replace select(). We are currently working to develop this.

Acknowledgments

We are grateful to Kathy Richardson of Digital's Network Systems Laboratory, for providing data on the performance of the Palo Alto Web proxies, and to Kathy and to Jessie Stickgold-Sarah for helping us evaluate our changes in the context of these proxies. We also thank the USENIX referees for their comments.

References

[1] J. Anderson, L. M. Berc, et al. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

[3] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Dec. 1997.

[4] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. To be presented at the Workshop on Internet Server Performance, June 1998.

[5] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[6] Digital UNIX Tuning Parameters for Web Servers. http://www.digital.com/info/internet/document/ias/tuning.html.

[7] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[8] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *1996 SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[9] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of the ACM Communications Architectures and Protocols Conference (SIGCOMM)*, pages 259–268, San Francisco, CA, Sept. 1993.

[10] A. Luotonen, H. F. Nielsen, and T. Berners-Lee. CERN httpd 3.0A. http://www.w3.org/pub/WWW/Daemon/, July 1996.

[11] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[12] R. E. McGrath. Performance of Several HTTP Demons on an HP 735 Workstation. http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html, Apr. 1995.

[13] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming tcp packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–280, Aug. 1993.

[14] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[15] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[16] Network Appliance, Inc., *NetCache*. http://www.netapp.com/level3/netcache/datasheet.html.

[17] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference. http://www.scriptics.com/people/john.ousterhout/threads.ps.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. Technical Report TR97-294, Rice University, CS Dept., Houston, TX, 1997.

[19] K. J. Richardson. Personal communication, 1997.

[20] S. E. Schechte and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/ stuart/Tarantula/FirstPaper.html.

[21] Solaris 2 TCP/IP. http://www.sun.com/sunsoft/solaris/networking/tcpip.html.

[22] Squid. http://squid.nlanr.net/Squid/.

[23] W. Stevens. *Unix Network Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[24] thttpd. http://www.acme.com/software/thttpd/.

[25] Zeus. http://www.zeus.co.uk/.

Tribeca: A System for Managing Large Databases of Network Traffic

Mark Sullivan
Juno Online Services
120 West 45th Street, 15th floor
New York, NY 10036
sullivan@staff.juno.com

Andrew Heybey
Niksun, Inc.
180 Tices Lane
East Brunswick, NJ 08816
ath@niksun.com

Abstract

The engineers who analyze traffic on high bandwidth networks must filter and aggregate either recorded traces of network packets or live traffic from the network itself. These engineers perform operations similar to database queries, but cannot use conventional data managers because of performance concerns and a semantic mismatch between the analysis operations and the operations supported by commercial DBMSs. Traffic analysis does not require fast random access, transactional update, or relational joins. Rather, it needs fast sequential access to a stream of traffic records and the ability to filter, aggregate, define windows, demultiplex, and remultiplex the stream.

Tribeca is an extensible, stream-oriented DBMS designed to support network traffic analysis. It combines ideas from temporal and sequence databases with an implementation optimized for databases stored on high speed ID-1 tapes or arriving in real time from the network. The paper describes Tribeca's query language, executor and optimizer as well as performance measurements of a prototype implementation.

1 Introduction

The rapid growth of high speed computer and telephone networks means that the tools to analyze and engineer the networks are becoming more and more important. Network engineers use a combination of hardware and software tools to monitor the network, record various statistics and flows, and analyze the collected data. These tools either operate directly on the live network or record traffic for later offline analysis. For example, one group we have worked with records OC3 links (155 Mb/s) and groups of 16 T1 links (1.5Mb/s each, 24Mb/s aggregate). Their tape technology ranges from 8mm tape to 96 GByte ID-1 tapes that transfer data at about 256 Mb/s. The data from a monitoring run ranges from a few gigabytes to hundreds of gigabytes. Network engineers expect this number to grow rapidly into the terabyte range as monitoring tools, networks, and storage technologies improve in price and performance.

Network traffic engineers use their vast collections of network data to perform such diverse tasks as protocol performance analysis, conformance testing, error monitoring and fraud detection. In general, each group writes its own ad-hoc programs to examine and analyze the data. Although these programs query large databases of recordings, the traffic engineers avoid using conventional relational database management systems (RDBMSs) for several reasons:

- Both the data and the storage medium are stream-oriented. Fast sequential access to data is crucial; transactional updates, fast access to random records, and concurrency control are not. A highly-tuned C program can outperform a general purpose RDBMS on this workload.
- RDBMSs do not usually handle data on tape well [3]. Non-clustered indices will not work for traffic data. Worse, traffic analysis data is often used only a few times (or once), so load time is a significant cost. Finally, network traffic traces contain many small records with fields a few bits wide, so per-tuple overheads can noticeably increase the database size.
- A network traffic trace is a sequence of time-stamped network protocol headers. The ana-

This work was done while both researchers were employees of Bell Communications Research.

lysts use operators like those found in sequence and temporal DBMSs [13][16]. Traffic analysts usually calculate aggregates on packet inter-arrival times or calculate and compare network utilization over successive time periods or time scales. The traffic applications also use several data-flow operators and pattern matching operators (e.g. demultiplexing and protocol recognition) that are not common in sequence databases.

- Traffic analysts run batches of related queries during a single pass over the data. Users will sometimes intentionally write queries that use partial results generated by a concurrently executing query. The shared single data source means that even otherwise distinct queries will often share subqueries.
- Users must consider the capacity of the analysis hardware. Often the users would rather reformulate an expensive query or drop an expensive query from the mix than overallocate processor or memory resources in the analysis machine. Relational systems run queries as fast as they can but typically do not provide this kind of capacity information.

Tribeca is a software system for monitoring and analyzing either a live network or recorded network traffic on tape. Tribeca users can write queries to process arbitrarily long streams of information. Like a relational DBMS, Tribeca has a query language that can be compiled and optimized. Like extensible DBMSs [15][5][2], Tribeca has a type system and user-defined operators so it can integrate support for different network protocols and specialized traffic analysis operators. Unlike conventional systems, Tribeca does not support random access to data, transactional updates, conventional indices, or traditional joins.

Tribeca is designed to read a stream of data from a single source (tape or a network interface) and apply compiled queries to the stream. It has a data-flow-oriented query language that allows users to construct large batch queries for the one pass over the data. It also has operations to separate and recombine substreams derived from the source. Finally, Tribeca supports window operators that allow users to compute moving aggregates and to do a very restricted form of join. Both the query language and the optimizer help prevent users from expressing queries that produce intermediate results that cannot be stored in main memory. Because of this, query optimization focuses on memory management and predicate ordering rather than traditional I/O optimizations like access path selection and join optimizations.

Several different groups of network analysts used Tribeca over a one-year period and the system performed well. Measurements show that it is only 1–9% slower than a hand-tuned ad-hoc program on simple queries. With Tribeca, our users are also able to construct more complex queries than they would be able to implement easily in their ad-hoc programs. More importantly, they can easily retarget their queries to do similar analysis on different kinds of networks.

This paper describes the Tribeca design and implementation. Section two gives an overview of the query language. Section three outlines the system's implementation and presents performance measurements from our prototype. Section four compares Tribeca to related work and section five gives conclusions.

2 The Tribeca Query Language

This section describes the syntax and semantics of the query language primitives. Implementation issues are deferred as much as possible until the next section. Before introducing the primitives, we present a motivating example.

2.1 Overview and Example

Figure 1 graphically shows a query used in characterizing IP-over-ATM traffic [7]. The analysts in this case look for "burstiness" in the packet arrival rates and changes in the distribution of packet lengths in order to help plan network capacity. They compare the characteristics of interest over several different time scales (ms, sec, min, hour). In order to isolate a bursty host, they group packet streams by source and destination, calculating similar aggregates over these groups.

The data set for the example is a traffic trace including Asynchronous Transfer Mode (ATM) cells from a dozen virtual circuits (VCs) multiplexed onto the monitored network link. Each trace record contains a time stamp and an ATM cell. The query takes a stream of ATM cells, discards those cells belonging to an uninteresting VC, demultiplexes the stream by

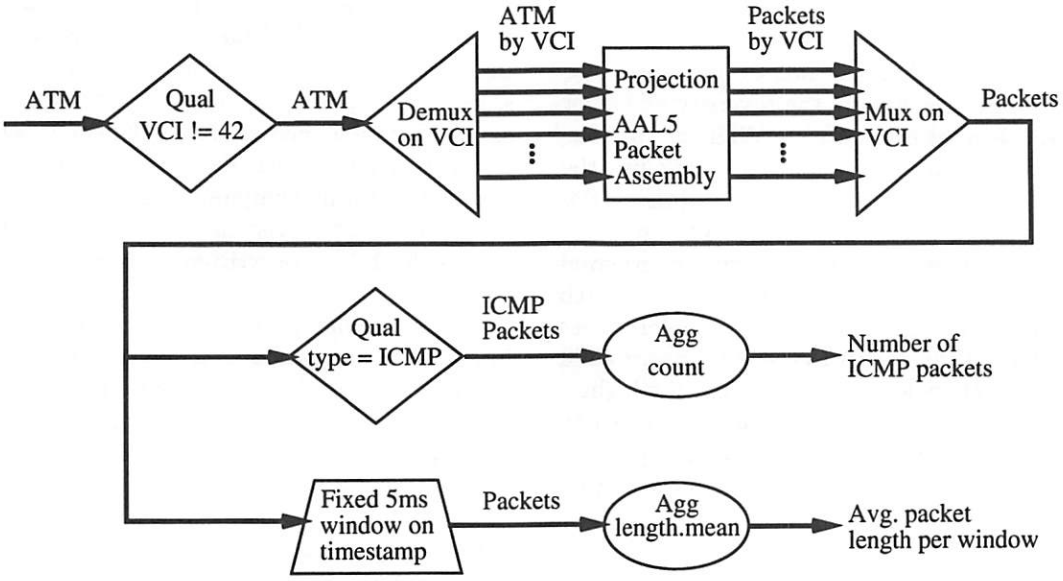


Figure 1: Graphical Representation of a Traffic Analysis Query

virtual circuit identifier (VCI), assembles IP packets (which are fragmented across successive cells on each VC), remultiplexes the packets back into a single stream, then counts the number of ICMP packets and finds the average length of all IP packets in each 5 ms interval.

The example shows several features of the query language presented in this section. Different layers of protocol are implemented by different data types. The example uses simple filters, aggregates, demultiplexing, multiplexing and some simple windows. Each of these is described in subsections below.

2.2 The Tribeca Type System

Tribeca has both a data description language (DDL) and an extensible type system. Like other extensible database managers, the core data management software is type-independent. Data types and operators may be added to the system to support new applications. The procedure for creating extensions in Tribeca is similar to that of Postgres [15] so the details are omitted here. An extension type declaration defines operators and data representations (ascii, host byte order, network byte order) associated with the type. Tribeca allows operator overloading so the same operator name can be used in different data types.

The DDL allows users to create composite types from the compiled-in extension types. The DDL has a simple inheritance mechanism that allows users to describe the kinds of layered packet headers that are commonly found in network traffic data (for instance, UDP/IP and TCP/IP types both inherit from the IP type). In addition to inheritance, the DDL has built-in support for arbitrary offset and width bit fields since network protocols often include bit fields. The DDL has an enumerated type provision so that queries can refer to ID fields by name instead of number (e.g. the field that determines that a routed frame relay packet is transporting IP data has the value 0xCC, but this value can be referred to in a Tribeca query as "IP"). Note that ad-hoc, unnamed composite types can also be created in queries.

For traffic analysis, support of both an extensible type system and a DDL are crucial. Extensions are needed because some hardware-generated time stamp fields and some fields of network protocols are difficult to describe in a data description language (for example, the DLCI field of a frame relay packet takes several bits from two different bytes of the packet header and combines them into a short int). Extensions are also used to incorporate into Tribeca the exotic statistical estimators used by the traffic analysts. The analysts also want control over the implementation of less exotic estimators, like mean, to ensure that the operator will be numerically stable for their workload. The DDL is important because it allows our non-programmer users to retarget their queries at new networks or at higher levels of the protocol stack without implementing

extensions.

The inheritance mechanism and ad-hoc types also help Tribeca queries handle the diversity of higher level protocols used in networks. While at the lowest level all packets on the same network use the same protocol (i.e. a frame relay network carries only frame relay packets), higher level protocols can be quite diverse and their packets interleaved in complex ways. A Tribeca query examines each packet to find out what higher level protocol it uses and then coerces the packet to the appropriate child type and extracts fields of interest from the higher-level part of the packet. This extraction creates an ad-hoc type that can be examined by later parts of the query. We'll see an example of this in the section on multiplexing.

2.3 Streams, Basic Operators, and Simple Queries

Every Tribeca query has a single *source* stream and at least one *result* stream. Stream declarations associate a name with the external source (result) of the stream data, such as a disk or tape file. The *source_stream* statement must also declare the stream's data type. The data types of result streams are derived from the operator that writes to them.

Tribeca supports three kinds of simple operators: qualifications, projections and aggregates. A Tribeca query can combine operators using pipes and transform the stream in several stages. Note that while Tribeca borrows the Unix term "pipe", these are not Unix pipes. Tribeca operators are *not* implemented as separate processes. As the example below illustrates, a Tribeca pipe statement names a stream of data and allows users to express data flow from one Tribeca operator to another.

Qualification operators filter data in a stream. Tribeca's qualification statement specifies a source stream, a result stream, and a list of qualification operators to be applied to the source. Records from the source stream that pass the qualifications are placed on the result stream. While the list of operators is implicitly a conjunction, Tribeca supports the usual complex qualifications involving AND, OR and NOT.

A projection selects one or more fields from each record in the source stream, assembles the fields into a new record, and puts the record onto the result stream. The projection statement may also apply a function to the field during the projection operation. It is important to realize that the projection statement is provided to allow users to construct simple, readable queries. As explained in Section 3, the stream data model allows most projections to be eliminated during compilation. Intermediate tuples are never materialized in Tribeca unless they are used as hash keys or written to external storage.

An aggregate operator is applied to all of the values in a stream and produces a single value. While aggregates of basic Tribeca streams are sometimes useful, queries usually produce streams of related aggregates using demultiplex and window operations described below.

The simple query below uses all operators introduced so far:

```
source_stream s1 is {tape sample1 AtmTrace}
result_stream r1 is {file res1}
result_stream r2 is {file res2}
stream_pipe p1 p2
stream_proj {{s1.atm.ts s1.atm.vci}} p1
stream_qual {{p1.ts.lte 1000000}} r1
stream_qual {{p1.vci.gt 5} {p1.vci.lt 50}} p2
stream_agg {p2.ts.min} r2
```

The query reads a source stream of type AtmTrace from tape. It uses project to create a stream of (time stamp, VCI) pairs (an ad-hoc composite type). That stream is then passed to two different quals. The first saves into a file all (ts,VCI) pairs with timestamp less than 1 million. The second finds all (ts,VCI) pairs in which the VCI is between 5 and 50. Finally, the aggregate finds the minimum time stamp from those pairs.

As described so far, Tribeca queries are trees of stream operations. The source stream may feed any number of operators. Each operator writes to a pipe or a result stream. The source and intermediate streams may feed any number of operators. An intermediate or result stream derives data type from the operator that writes it. None of the simple stream operations introduced so far take their input from more than one stream, but we will introduce operators for combining streams in the next few sections.

2.4 Demultiplexing and Remultiplexing Streams

Traffic analysis queries often must partition a stream into substreams, process the substreams, then recombine the results of the substream analysis. The demultiplexing operation partitions records in a stream based on data in each record. The query below partitions *s1* into substreams of ATM trace records based on virtual circuits, then finds the max time stamp for each VCI:

```
stream_demux {s1.atm.vci} p1
stream_agg {p1.ts.max} p2
```

P1 is not a single stream but a set of substreams, each with the cells for one virtual circuit. The pipe *p2* represents a collection of logical substreams each containing the max time stamp for one VCI.

In the example, demux is used like a groupby operator in a relational DBMS. However, the demultiplex operator allows users to apply a series of operations to the demultiplexed streams instead of simply applying aggregates. We can, for example, demultiplex ATM cells by VCI, assemble IP packets from consecutive cell payloads, then apply an aggregate to the IP stream.

In the example below, the query first divides the stream into virtual circuits (*p1*). Each logical substream on *p1* is a sequence of ATM cells for a distinct VCI. Next, consecutive cell payloads from the same circuit are assembled into a stream of IP packets associated with that circuit (usually there are several cells per IP packet. Assembly is actually more complex than this and is described in detail in the subsection on Windows). Finally, the IP stream is qualified and an aggregate is applied to each qualified TCP/IP substream.

```
stream_demux {s1.atm.vci} p1
stream_proj {{p1.assemble_ip}} p2
stream_qual {{p2.ip_type.eq TCP}} p3
stream_agg {{p3.atm.vci p3.count}} p4
```

P4 is a set of logical substreams each containing the count of TCP packets for one virtual circuit.

Tribeca allows users to demultiplex the same stream more than once. A second demux simply divides the original stream into more substreams. For instance, we can demux once by VCI then, after assembling IP packets, demux again by IP type (UDP, TCP, etc.). These operations produce substreams that are distinct for each VCI/ip_type pair

In partitioning the stream, the demux operator also "names" each substream. The substream name is not part of the data stream, but may be referred to in project and aggregate operations. In the example above, *p3* is a set of substreams containing $< VCI, count >$ pairs.

Unfortunately, the demultiplex operator allows users to express queries that cannot be executed in available memory. The demux implementation uses memory space proportional to the cardinality of the demux field. In practice, however, the number of distinct VCIs, packet types and so on in our data is small. Instead of removing demux from the language, Tribeca uses statistics and capacity planning support to help users avoid it when inappropriate. For traffic analysis, demux only really breaks down when users want to partition the packet stream into substreams by time stamp. Streams are long and are typically sorted by time stamp. Users often want to apply aggregates to packets grouped by time value. Tribeca's window feature (described below) allows users to group records by sort field in an efficient way.

2.4.1 Using Multiplex to Combine Streams

The multiplexing primitive is a simple operator for efficiently combining Tribeca streams. It can be used in two ways. In the first usage, mux is used to combine the logical substreams produced by demux. In the second usage, mux combines unrelated streams of the same data type.

Below is an example of the first usage of the multiplex operator. The query uses demux to divide the stream by VCI and counts the packets on each VC. The ⟨ VCI, count ⟩ pairs are then combined into a single stream that is qualified and aggregated.

```
stream_demux {s1.atm.vci} p1
stream_agg {p1.atm.vci p1.count} p2
stream_mux p2 p3
stream_qual {p3.count.lt 100} p4
stream_agg {p4.count.mean} r1
```

Note that the mean applied after the mux operates on *all* ⟨ VCI, count ⟩ pairs with count greater than one hundred. Without the mux, the aggregate would have been applied to each virtual circuit separately.

If the stream is demultiplexed more than once, an optional argument to stream_mux allows the substreams to be partially recombined. Suppose a query demultiplexed an ATM cell stream by VCI, assembled it into IP packets (an aggregate), then demultiplexed it again by ip_type. The resulting stream is logically separated by both VCI and ip_type. Muxing by VCI, would leave a stream that was logically divided by only ip_type.

The second kind of multiplex operation, in which several streams are combined, is very common in traffic analysis. Often, several different combinations of the same high level and low level protocols are used in the same network. For instance, frame relay networks have many ways of transporting IP packets (routed, ethernet bridge, fddi bridge, etc.). The query below is a (much simplified) typical stage of frame relay analysis. It finds two types of IP packets, extracts the same interesting fields from both and then combines them into a single stream.

```
stream_qual {s1.is_routed_ip} p1
stream_proj {p1.ts p1.ip_type p1.ip_len} p2
stream_qual {s1.is_bridged_ip} p3
stream_proj {p3.ts p3.ip_type p3.ip_len} p4
stream_mux {p2 p4} p5
```

The output stream is a triple ⟨ time stamp, ip type, length ⟩.

2.5 Windows on Streams

The Tribeca window operator groups successive records in a stream so they may be operated on as a unit. Tribeca supports two kinds of windows. A *fixed* window is effectively a demultiplex operation for the stream's sort field. (Network traffic traces are sorted by time, so the sort field is usually a time stamp). It partitions the stream into non-overlapping groups of records. A *moving* window breaks the stream into successive overlapping groups of records. Each one is illustrated in the example below. Result *r1* contains the number and mean length of large (>100 byte) packets in successive five ms intervals. *R2* contains the inter-arrival time between successive packets:

```
stream_window w1 on s1
  defined by {s1.ts.interval 0.005} is fixed
stream_qual {w1.length.gt 100} p1
stream_agg {p1.count p1.length.mean } r1
stream_window w2 on s1
  defined by {s1.count 2} is moving
stream_agg {w2.ts.diff} r2
```

As the example shows, a Tribeca user-defined function delineates each window. The function may be applied either to the sort field value (time interval) or to the record's position in the stream (count). The window names, *w1* and *w2*, can be used as input to other Tribeca operators in the same way as Tribeca pipes.

In traffic analysis, aggregates are almost always used in conjunction with windows. After every windowfull of data, downstream aggregates produce values and are reinitialized for the next window. Also, packet assembly in ATM analysis is actually implemented using a combination of a window and an aggregate function. The window function is a predicate that returns TRUE on the ATM cell that contains the last byte of the IP packet. The aggregate is a function that combines ATM cell payloads into a single IP packet.

2.5.1 Window Filters

Tribeca streams are like RDBMS relations with one important restriction: Tribeca streams may not be joined. Arbitrary comparisons between records in streams are not allowed for performance reasons. A join operator would permit users to express queries that could be executed only by sorting or repeatedly passing over the data on tape.

The Tribeca *window filter*, however, is a restricted form of join operation that relates records in a window to records in a stream. At any given moment during the execution of a query, the records in a window can be compared to the current record in any stream. For example, the query below searches for cells whose VCI has appeared in a recent predecessor cell. It defines a moving window that records the last 100 atm cells seen, then compares the VCI of the current cell to those of the cells in the window.

```
stream_proj {s1.atm.vci} p1
stream_window w1 on p1
  defined by {s1.count 100} is moving
window_filter {{s1.atm.vci.eq w1}}
  {s1.vci s1.ts w1.ts} r1
```

The window filter arguments are a list of join predicates and a list of projected fields.

While users cannot compare arbitrary records in streams, window filters allow users to compare records that are temporally near one another. Window filters are not as powerful as true joins, but are useful in traffic analysis and can be executed efficiently as long as the window is small enough to fit in memory. For example, one traffic analysis query takes the mean and standard deviation of packet length over N seconds, then uses a window filter to search for packets that are significantly larger than the mean over the next N seconds. We also allow users to initialize windows from files so they can do things like select UDP packets destined for port numbers listed in a file.

3 Implementation and Optimization Issues

While Tribeca's query language and basic data path differ from a conventional DBMS, much of Tribeca's basic software architecture is very conventional. At run time, Tribeca compiles queries into a directed acyclic graph. Each node in the graph contains a list of predicates (like a RDBMS where clause) and a list of project/aggregate operations (like RDBMS target lists). A pipeline is a pointer connecting two nodes. The optimizer rewrites this graph to improve performance. The executor uses it to guide query execution. In the subsections below, we describe some of the Tribeca implementation and optimization strategies.

3.1 Basic Data Management

The stream data model allows Tribeca to perform its basic I/O operations efficiently. I/O in Tribeca consists only of reading the source stream and writing result streams. By design, joins do not affect I/O performance because one join operand must fit in memory. The large sequential read is the dominant I/O cost.

This simple I/O pattern means that Tribeca can take advantage of standard operating system services in a way that a conventional RDBMS cannot. Modern Unix file systems are tuned to detect when applications are doing large sequential reads and support them efficiently. Because a conventional RDBMS accesses data differently from a standard application program, the conventional DBMS must work around operating system resource managers such as the file system and buffer pool. Also, because Tribeca queries do not update data in place, Tribeca needs none of the support and overhead required for concurrency control.

Because of its mix of workloads, a conventional RDBMS faces page size tradeoffs that Tribeca does not; Tribeca has no fixed page size. At run time, it divides available I/O buffer space among its source and destination streams according to the expected volume on those streams (i.e. the source stream is usually very large and the others are small). All streams are double-buffered so that one buffer can be processed while another is being read or written. All read and write operations are in units of full buffers.

Unlike RDBMS's and many other programs, Tribeca cannot preformat its data. That means that it must separate the source stream into records as the data is processed (relational systems use a page structure so repeated scans of the same data do not require record parsing). Tribeca supports record parsing strategies for three different kinds of data: fixed-size records, variable-size records, and "framed" records. The stream data type tells the Tribeca executor which strategy to use. "Framed" streams come from some of the high speed data recorders. Valid records in these streams contain framing patterns that are used to distinguish valid records from periodic bursts of "noise" bytes.

The last important data management issue is that Tribeca makes every effort to minimize internal data copying. Intermediate records produced by Tribeca queries are never unnecessarily materialized. Instead, a pointer to the original field value in the input buffer is used every place the projected value appears as an argument. Materialization occurs only if (a) data is copied to an output buffer (b) data must be aligned correctly for some operator (c) non-contiguous values have to be assembled for a hash key. Also, note that projections involving user-defined functions, cannot normally be eliminated by compilation.

Multiplex and projection operators can often be removed at compilation, sometimes with the help of an extra level of indirection. For example, in frame relay, IP packet headers can appear at different offsets within the packet (depending on whether the frame relay packet is routed or bridged, for instance). Operators downstream from the multiplex must use extra indirection, but the operator does not involve copying.

3.2 Using Coarse-Grain Indices

While Tribeca is largely designed so that a stream of data coming from the network and a stream of data coming from tape can be manipulated in the same way, there is one important difference between the two cases: secondary indices can be constructed for recorded data. Traffic analysis users often identify a few hours of packets that are especially interesting and issue repeated queries over this data. Secondary indices are very important for this workload.

When the database is stored on large, high-speed tapes, however, only a limited kind of index is feasible. Our ID-1 tapes are simply not effective random access devices. That means that we cannot support indices on non-sort fields. Even on the sort field, a full index of every packet in the stream is impractical. The index itself cannot be stored on tape, so it must be much smaller than the data.

Therefore, Tribeca supports what we call *coarse-grained secondary indices.* A coarse-grained index is an approximate index on the sort field of recorded data streams. Users specify the ratio, R, of index size to underlying data size. When building an index, Tribeca inserts a key pointing to one record for every R bytes. The index is always stored on disk.

After the index is constructed, a subsequent query with a qualification on the indexed sort field can use the index to skip over parts of the input data. Because the index is approximate, the scan cursor must be placed on the last indexed record before the scan key. After that, Tribeca applies the qualification to each record until the matching one is found. The search procedure is not as fast as a full index, but it represents a good compromise for the traffic analysis environment. Users can trade off search performance and the size of the index (larger R means smaller indices but a coarser grain search).

3.3 Implementing Fixed and Moving Windows

Tribeca implements fixed and moving windows in different ways. Often, fixed windows require no buffering. If the fixed window is not used in a filter, records are processed through the downstream executor nodes as soon as they arrive in the window. When the fixed window is flushed, Tribeca walks the executor nodes below the window generating aggregate results and reinitializing aggregate nodes for the next window.

A moving window is implemented as a circular buffer. If several windows overlap, the largest window determines the size of the circular buffer. The other windows are represented by pointers into the buffer required by the larger window. Like SEQ [14], Tribeca allows extension implementors to define *moving aggregate functions.* A normal aggregate function takes a single record as an argument (the "current" record in the stream). A window flush causes the normal aggregate function to be applied once for every record in the window. A moving aggregate function is called once per window flush and takes as arguments pointers to the records entering and leaving the window.

Tribeca includes two implementations of window filter: one based on hashing and the other on nested loop. In the first case, the window contents are used to build a hash table and the stream argument probes the hash table. In the second, Tribeca iterates over the window contents for each element in the stream. The hash table is more effective if the window is large and relatively stable. For small windows, the cost of creating and destroying the hash table overrides the cost of iterating over the window contents.

3.4 Optimization Issues

Query optimization in Tribeca has two competing goals. The first is to minimize query execution time. The second is to ensure that the intermediate state associated with the query fits in main memory. Tribeca approaches the first goal like a standard RDBMS. For the second goal, it must pay special attention to two Tribeca operators because they require a data-dependent amount of memory. First, a demux operator is implemented as a hash table whose size depends on the cardinality of the demultiplex field. Second, moving windows and window

filters require buffer space to hold their contents. The buffer is data-dependent if the window size is a function of the sort field.

In both space- and time-based optimization, Tribeca benefits from some standard RDBMS optimizations. As is often the case in an RDBMS, pushing qualifications towards the source of the data is helpful so Tribeca migrates quals towards the source stream. Successive qualifications are then grouped together so that the optimizer can use constraint minimization to eliminate redundant quals and order the predicates to minimize expected cost. Migration is, of course, not always possible. For instance, a qual cannot move past an aggregate operator and cannot be moved past a positional window. Tribeca also does some simple common sub-expression elimination. If two instances of the same operator have the same input, they are combined. It should eventually be possible to combine and eliminate larger sequences of operators.

Qual migration can help reduce the number of tuples in windows and reduce the cardinality of demuxes. To further minimize the storage cost of windows that require buffering, Tribeca does several things. First, it combines overlapping windows when it can. Second, it migrates any functional projections whose inputs are smaller than their outputs in front of the window. Still, overflow can occur. On overflow, Tribeca drops records entering the full buffer rather than allow disk I/O. Better user control of overflow error handling is an area for future work.

3.5 Performance Measurements

We measured Tribeca on a Sun Sparc 10 performing queries on two datasets. One data set consisted of frame relay traffic (carrying mostly IP), and the other was classical IP-over-ATM [7] traffic. We used data stored on disk (using the standard SunOS UFS filesystem) and ID-1 tape to perform our measurements. The measurements were run on 260 megabyte disk files and a 10 gigabyte tape file. All tests were run in single-user mode to prevent other user activity from affecting the test results.

We ran a variety of queries of increasing complexity to measure Tribeca's performance as its tasks became more compute-intensive. The queries exercised most of Tribeca's features. Table 1 describes the queries run and the measurement results for Tribeca running on 260 megabyte disk files. In the table the results are given as kilobytes per second and records per second. For ATM the records are 64 bytes long while for frame relay the records are variable length. As a baseline, we also present the speed at which the Unix `dd` program can read the disk file. The `dd` program simply reads the file in blocks whose size is specified by the user (we used the same block size for `dd`, the stand-alone programs and Tribeca), and then writes to the Unix null device (`/dev/null`).

Tribeca compares favorably to the baseline for all of the frame relay measurements, ranging from 1.3 to 5.8 percent slower. The speed decrease is because Tribeca must perform some processing on each record in the file while `dd` does not touch the data at all. When processing ATM cells, Tribeca is considerably slower than the baseline. The records in the ATM data set are much smaller than the frame relay records and the ATM cell record format includes several bit fields that must be reconstructed to perform the query. Tribeca thus spends correspondingly more time processing each record. In these tests Tribeca used 70–75% of the workstation's CPU while `dd` used about 68%.

We estimated the overhead introduced by Tribeca by comparing its performance on several sample queries (`frame1` through `frame4`) to that of a simple stand-alone C program performing the same functionality. The stand-alone program is hard-coded to perform only the tested query so it does not have any of the overhead required to execute a general-purpose query. Table 2 gives the comparison between Tribeca and the stand-alone programs. The table shows that Tribeca introduces no more than 5% overhead relative to the stand-alone program in the test queries. The stand-alone program used about 5% less CPU time than Tribeca.

Finally, Table 3 compares the performance of Tribeca and a stand-alone program when running from a relatively large (10 gigabyte) dataset on ID-1 tape. We only compared Tribeca to a stand-alone program on two queries because of the time required to run the tests. In this case Tribeca ran between 5 and 9 percent slower than the stand-alone program. Both Tribeca and the stand-alone programs used about 98% of the CPU in these tests. The percent CPU used is much higher in the tape tests because the HiPPI interface used to connect to the ID-1 tape drive uses programmed I/O. The device driver must copy each word of data coming from the

Query	Description	Records/sec	KBytes/sec
`dd`	The Unix `dd` program reading the disk file.	N/A	5754
`frame1`	Count all the records in the frame relay trace and sum the length fields of each record.	19130	5423
`frame2`	As `frame1` but qualify on the message type of the record (the message type is the high four bits of a byte in the header).	19992	5667
`frame3`	As `frame2` but further qualify on two character fields (the control and nlpid fields of the frame relay header).	19992	5667
`frame4`	As `frame3` but qualify on two more single-byte fields, the T1 interface and channel (the recorder has multiple interface boards, each board has multiple T1 lines and each line can have multiple channels).	20021	5675
`frame5`	Qualify to select only IP packets, then demux by source and destination port and count and sum lengths.	19968	5660
`frame6`	Qualify to select only IP, then demux by T1 board, interface and channel. Count the packets and sum the lengths on each group.	20021	5675
`frame7`	Qualify by message type and protocol to get two streams of IP packets in slightly different formats (frame relay can carry IP in several different formats). Project the IP protocol field and packet length from each stream and multiplex together to count the total number of IP packets and bytes (regardless of the format used).	19764	5602
`frame8`	As `frame7` but include the board and channel in the projection and then demux by board and channel to count IP packets on each channel.	19816	5617
`atm1`	Demux by VCI and count cells in each.	50137	3133
`atm2`	Demux by VCI/VPI and count cells in each.	45098	2818
`atm5`	Qualify to select a particular VCI and count cells.	76746	4796
`atm6`	Qualify to select a particular VCI/VPI and count cells.	77425	4839

Table 1: Performance Results for Queries run on Disk

tape. For these tests over 95% of the CPU time is system time.

Our measurements demonstrate that Tribeca adds (in the worst case) 9% more processing overhead than a special purpose program tailored to perform the same query. In most of the tested queries, Tribeca performed even better. The small cost is far outweighed by the flexibility and convenience of changing small simple queries rather than re-writing C code to perform different analyses.

4 Related Work

The difficulties in using relational databases stored on tape are overviewed in [3]. Sarawagi [11] modifies a relational query optimizer to consider large tape archives in its cost formula and caches tape data on faster storage. Video-on-demand systems [4] might use tape storage, but in these workloads many users randomly access independent large objects instead of sequences of small ones.

A temporal DBMS usually treats time as an additional dimension [16]. Implementation issues involve multidimensional indices [6] and disk-based temporal joins [8]. Network traces are temporal, but Tribeca treats it as a one-dimensional stream. Also, so far, the traffic analysts have treated data as events rather than intervals, making temporal joins simpler in Tribeca. Illustra [17] implements time-series data as an ADT, but does not have operators like demux required for traffic analysis.

The SEQ sequence DBMS [12][13][14] integrates sequence operations into an RDBMS. It has operators analogous to those defined in Tribeca although the data flow style of Tribeca's query language should make constructing large batches of sequence queries

Query	Tribeca		Stand-alone program		Ratio
	Records/sec	KBytes/sec	Records/sec	KBytes/sec	
`frame1`	19130	5423	20151	5712	0.95
`frame2`	19992	5667	20312	5757	0.98
`frame3`	19992	5667	20256	5742	0.99
`frame4`	20021	5675	20363	5772	0.98

Table 2: Tribeca Compared to Stand-alone Programs. The queries are as described in Table 1. The "Ratio" column is the ratio of Tribeca's performance to that of the stand-alone program.

Query	Tribeca		Stand-alone program		Ratio
	Records/sec	KBytes/sec	Records/sec	KBytes/sec	
`frame1`	27638	6427	30296	7045	0.91
`frame4`	28944	6730	30412	7072	0.95

Table 3: Tribeca Compared to Stand-alone Program on ID-1 Tape. The queries are as described in Table 1. The "Ratio" column is the ratio of Tribeca's performance to that of the stand-alone program.

easier. Because Tribeca eliminates features of the RDBMS that would slow sequence queries, it runs considerably faster than SEQ on sequence data. However, Tribeca will not support queries that mix relational and sequence data as SEQ does. Also, many of the SEQ optimization strategies involve teaching the relational optimizer to distinguish sequences from relations so the executor can access and buffer them differently. Because it does not support relations, Tribeca's optimizer does not face these problems. For example, Tribeca implements a window primitive instead of teaching the optimizer to distinguish a relation joined to itself from a window scan.

The Tangram system [10] implemented in Prolog has operators similar to Tribeca's basic stream processing operators. Tangram did not include more complex operators such as demux, mux, window, and window filter. Tangram was important early work in stream processing. However, processing stream data in a Prolog system has some potential performance drawbacks. The rule processing in Prolog systems is typically made efficient by carefully-tuned main memory data structures; they do not handle data on secondary or tertiary storage well. Recognizing this, the Tangram project used a relational system as a front-end to handle the bulk of the I/O processing and filtering for the prolog backend. Still, implementing both data management and stream processing together in the same engine will reduce data handling overheads. It will also allow users to write queries in a single query language instead of composing them partially in SQL and partially in Prolog. Further, as in SEQ, trying to implement two kinds of systems in one program will inevitably lead to performance compromises.

There have been several efforts at querying live networks. Datacycle [1] used a specialized network interface to query data circulating through a high speed local network. The Berkeley packet filter [9] allows users to load simple filters into the operating system kernel to generate qualified packet traces efficiently.

5 Conclusions

Network traffic analysis is used in a variety of applications ranging from performance analysis and network provisioning to fraud detection. The characteristics of the data sets and the analyses to be performed on them do not lend themselves to the use of a conventional DBMS, but writing custom programs for each query performed on the data is not desirable either. The data sets are extremely large (the IP-over-ATM trace mentioned earlier in the paper consists of approximately 176 GBytes of data) and come pre-ordered by timestamp. The only practical way to cope with the data is to either analyze it in real time as it happens or to record it on tape.

Tribeca is a stream-oriented database management system designed to support network traffic analysis. Its query language has a data flow character that is familiar to network analysts and supports sequence operators they use in their work. Tribeca's executor is tuned for sequential I/O and the optimizer is focused toward memory and processor lim-

itations rather than join ordering and access path selection. The current implementation of Tribeca has performed useful analyses on several large data sets. The overhead introduced by Tribeca relative to a special-purpose analysis program is not very large, i.e. the convenience and flexibility of Tribeca provide more than enough incentive for analysts to use it.

Acknowledgments

We would like to thank Yatin Saraiya, Sid Devadhur, Paul England, Daniel Barbara, Parag Pruthi, Walter Willinger, Bob Sherman, Namon Jackson, Andy Ogielski, Ravi Jain, and Debbie Swayne.

References

[1] T. Bowen et al. The Datacycle Architecture. *Communications of the ACM*, 35(2), December 1992.

[2] M. Carey et al. Object and file management in the exodus extensible database system. In *Proc. 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[3] M. Carey, L. Haas, and M. Livny. Tapes hold data, too: Challenges of tuples on tertiary store. In *Proc. ACM SIGMOD Conference*, 1993.

[4] A. Chervenak, D. Patterson, and R. Katz. Storage systems for movies-on-demand video services. In *IEEE Mass Storage Symposium*, 1995.

[5] L. Haas et al. Starburst midflight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[6] C. Kolovson. Indexing techniques for historical databases. In *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings Publishing Co., 1993.

[7] M. Laubach. Classical IP and ARP over ATM. Request for Comments 1577, Internet Engineering Task Force, January 1994.

[8] C. Leung and R. Muntz. Query processing for temporal databases. In *Proc. IEEE Conference on Data Engineering*, 1990.

[9] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter USENIX Technical Conference*, Jan 1993. San Diego.

[10] D. S. Parker. *Stream Data Analysis in Prolog*, chapter 8. MIT Press, 1990.

[11] S. Sarawagi. Query processing in tertiary memory databases. In *Proc. Conference on Very Large Data Bases*, 1995.

[12] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. ACM SIGMOD Conference*, 1994.

[13] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *Proc. IEEE Conference on Data Engineering*, 1995.

[14] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proc. 1996 VLDB Conference*, Mumbai, India, September 1996.

[15] M. Stonebraker, L. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[16] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segeve, and R. Snodgrass, editors. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings Publishing Co., 1993.

[17] Illustra Information Technologies. Illustra timeseries data blade. Information available via HTTP from www.illustra.com.

Transparent Result Caching*

Amin Vahdat
Computer Science Division
University of California, Berkeley

Thomas Anderson
Department of Computer Science and Engineering
University of Washington, Seattle

Abstract

The goal of this work is to develop a general framework for transparently managing the interactions and dependencies among input files, development tools, and output files. By unobtrusively monitoring the execution of unmodified programs, we are able to track *process lineage*—each process's parent, children, input files, and output files, and *file dependency*—for each file, the sequence of operations and the set of input files used to create the file. We use this information to implement Transparent Result Caching (TREC) and describe how TREC is used to build a number of useful user utilities. *Unmake* allows users to query TREC for file lineage information, including the full sequence of programs executed to create a particular output file. *Transparent Make* uses TREC to automatically generate dependency information by observing program execution, freeing end users from the need to explicitly specify dependency information (i.e., Makefiles can be replaced by shell scripts). *Dynamic Web Object Caching* allows for the caching of certain dynamically generated web pages, improving server performance and client latency.

1 Introduction

The goal of this work is to develop a general framework for transparently managing the interactions and dependencies among input files, development tools, and output files. By unobtrusively monitoring the execution of unmodified programs, we are able to track *process lineage*—each process's parent, children, input files, and output files, and *file dependency*—for each file, the sequence of operations and the set of input files used to create the file. This information can be used to determine the exact sequence of operations used to create any system file or to keep the contents of output files synchronized as dependent input files are modified.

As a motivating example, several years ago, it was discovered that some published satellite data had been incorrectly normalized; however, because of the lack of software support, it has been difficult to identify exactly which experimental results were tainted by the error. It is believed that several journal articles have since been published still based on the incorrect data [Dozier 1993]. As another example, one error common to program developers is introducing a new header file without manually updating dependency information. This can result in an executable with object files based on different versions of the same header file, often resulting in subtle bugs where different modules reading and writing the same fields of a data structure in fact access different regions of memory.

The combination of transparently obtaining process lineage and file dependency information provides a powerful substrate for developing applications in a wide range of application domains:

- *Unmake:* Unmake allows users to query the system for file lineage information, including the full sequence of processes executed to create a particular output file. For example, users running simulations who neglect to document the specific parameters used to generate output files can query Un-

*This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Novell, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was supported by a National Science Foundation Presidential Faculty Fellowship. For more information, send email to vahdat@cs.berkeley.edu.

make for the program used to create the output, the specific command line parameters, and the environment variables in effect when the command was executed.

- *Accountability and Access Control:* Related to the Unmake application, TREC can be used to perform logging of programs run as `root`. System administrators are often forced to give `root` privileges to multiple users. Unfortunately, this means that accountability is sacrificed, making it difficult to ascertain the identity of individuals responsible for particular actions. Since Unmake can provide process lineage information, it can trace file accesses back to the shell (and hence the user id) of the process that originally executed `su`. Such a tool can also be extended to monitor system calls, disallowing certain accesses based on the "effective" `uid` of the calling process [Goldberg et al. 1996]. For example, Bob acting as root may be disallowed write access to all files in `/dev` and `/etc`.

- *Transparent Make:* This version of the `make` utility allows users to specify the sequence of operations for constructing output files as simple shell scripts. The first time the shell script is run, TREC determines the set of files that affect output files through empirical observation. During subsequent executions of the shell script, TREC can re-run only those commands that have been invalidated by changes to input files. This approach has two advantages. First, it frees users from manually specifying dependency information in a language that can be restrictive [Levin & McJones 1993]. Next, transparent make does not require users to manually update dependency information. Thus, when a new header file is added to a source tree, TREC transparently adds the new dependency to its lineage information by observing the inputs to subsequent compilations. Similarly, if a tool's command line parameters must continuously be updated to produce output files, TREC automatically matches each output file to the parameters used to generate it.

- *Dynamic Web Object Caching:* Today many web pages are constructed dynamically as a result of user input. One example in the web today is using CGI-bin programs to produce HTML pages. For instance, to download the latest version of Netscape's Navigator, users answer a number of questions about their platform before being presented with a page enumerating URLs for the correct binary. The disadvantage of using CGI-bin programs are: (i) servers, proxies, and clients are normally unable to cache the page because the contents might change for every invocation of the program, and (ii) retrieving such content incurs extra overhead at servers because a program must be run to generate the content (as opposed to transmitting an existing file). Given some locality in user input, it would be cheaper to cache the results of CGI-bin program execution with popular input patterns (e.g., users wishing to download the latest English version of Navigator for Windows95/NT), reducing both server load and client latency. While existing work allows for applications to be written that can cache their results [Iyenger & Challenger 1997], TREC automates this process by automatically caching program results and invalidates such results when the input to a CGI program changes (e.g., a new version of Navigator becomes available). This application is more active than the previous two examples: TREC dependency information is used to generate specific actions whenever a pre-specified operation takes place (an output file is invalidated when its input files are modified).

In this paper, we demonstrate how our prototype framework for *Transparent Result Caching* (TREC) is used to implement three of the above applications: unmake, transparent make, and dynamic web object caching.

TREC captures file dependency information by intercepting a small number of system calls using native kernel tracing mechanisms. Using the information from these calls, TREC maintains the following information for each process: command line arguments, environment variables, process parent, process children, files read (input files), and files written (output files). TREC then organizes this information hierarchically, allowing users to query the system for file lineage, for example, to determine all processes involved in creating an output file. With transparent make for example, we are able to automatically determine the set of include files associated with any compilation by observing the I/O behavior of the compiler. Thus, when an include file is modified, transparent make automatically determines that recompilation is necessary from process lineage information. Relative to existing techniques for automatically determining dependencies, our approach has the added benefit of not requiring a separate parser for each different programming language syntax.

In contrast to our approach using TREC, `make` and related software configuration management tools are currently used to specify and maintain dependency information. Such tools suffer from a number of deficiencies. For example, to manage file and program dependencies,

users must manually specify dependency information. Programmers must specify the dependencies between source files, object files, and executables. If any changes occur, such as introducing a new header file, users must remember to manually update the dependency information. With TREC, maintaining dependency information is both simpler and less error-prone because dependencies are deduced transparently by observing program execution.

Another shortcoming of `make` and related tools is the inability to track file lineage. Makefiles only implicitly contain lineage information; if the Makefile changes, the lineage information about existing output files can also be destroyed. As described above, lineage information is helpful in a number of contexts. If an output file does not contain expected results, debugging is easiest by working backwards to see whether the problem is with the file inputs, the data analysis tool, or the command line parameters. Similarly, if an input file is discovered to have a flaw, it is helpful to know all the output files derived from the input.

Finally, `make` is largely targeted toward software development; it can be too static to be useful for other communities. For instance, scientists often spend their time exploring different sequences of tools, different parameters, and different parts of an image. For example, one tool might extract the pixel values for a latitude and longitude region from a set of files containing satellite images. However, the images can overlap, and the requested region may span multiple image files. The input files actually read to create an output file can vary depending on the command line parameters passed to the tool. Expressing such dynamic dependencies can be difficult with Makefiles. TREC, on the other hand, is well-suited for managing dynamic dependencies because of its ability to discern file lineage simply by observing program execution.

The rest of this paper is organized as follows. Our implementation of TREC, its baseline performance, and limitations of TREC profiling are described in detail in Section 2. In Section 3, we describe how TREC is used to implement our three sample applications, unmake, transparent make, and dynamic web object caching. Section 4 describes related work and Section 5 presents our conclusions.

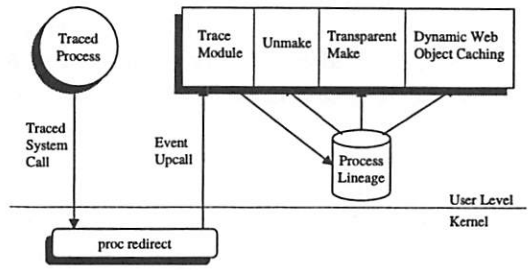


Figure 1: This figure describes the TREC architecture.

2 Implementation

2.1 Architecture

TREC is implemented using the Solaris `proc` file system (most major UNIX variants provide a substantially similar interface) to intercept the set of system calls necessary to build dependency information. For our target UNIX architecture this set includes the following system calls: `open`, `fork`, `fork1`, `creat`, `unlink`, `exec`, `execve`, `rename` and `exit`. By catching these system calls, TREC is able to determine full process lineage information. Command line parameters and environment variables are available from the `exec` system calls. TREC determines the set of input files and output files by examining the options to the `open` system call (targets of `creat` are assumed to be output files). This design choice potentially over-constrains the set of input and output files because, for example, a file opened for writing that is not actually written to with a `write` system call is considered an output file. We made this design decision because of the added overhead of intercepting all `read` and `write` system calls in addition to the set of calls already being monitored. For some programs, we observed that `read` and `write` operations were executed four times as often as all other traced system calls combined. Note that earlier file system tracing studies also inferred read and write operations to avoid the extra overhead of tracing the read/write system calls [Baker et al. 1991].

The overall TREC system architecture is summarized in Figure 1. TREC runs as a multi-threaded process that attaches to a target process using the `proc` file system interface. A trace thread is responsible for building lineage information. Other threads use this lineage information to implement the higher-level services described in Section 3. Given a list of target system calls,

`proc` forces a context switch to the TREC tracing thread whenever a relevant call is executed by the attached process or any of its children. The tracing module exports a callback-based interface to application modules. Modules, such as transparent make, use the callbacks to determine when output files are modified. Thus, when a callback is received that a file is modified, a module can take action on dependent output files—invalidating the output or re-generating it for example. Currently, callbacks are exported for file read and write events.

TREC uses the system call information to build a lineage tree of the target process and all of its children. Each node of the lineage tree represents an executed process and contains the following information: execution time, command line arguments, environment variables, files read, and files written (a file that is both read and written by a program is considered to be only an output file). An example of this lineage information is presented in Section 3.1.

2.2 Performance

Since the context switches imposed by the `proc` file system required to perform our tracing can impose significant overhead, we took a number of measurements to quantify the slowdown. Table 1 quantifies the TREC overhead for three simple benchmarks. All benchmarks were conducted on a 167 Mhz Sun Ultra/1 workstation running Solaris 2.5.1. The first benchmark, open, calls `open` and `close` on the same file in a tight loop 5000 times (note that only the open call is actually traced). While the 54.8% overhead imposed by TREC is significant, the next two benchmarks demonstrate the slowdown of individual system calls do not adversely affect the performance of real applications. The next benchmark is a compilation of the Apache HTTP server, version 1.2.4 [Apa 1995]. The source tree consists of 38,000 lines of C code and was compiled over NFS. While the 13.9% slowdown is noticeable, we believe it to be tolerable. The final benchmark, Latex, involved running `latex` four times, `bibtex`, and finally `dvips` to produce postscript for a 17 page document. For this benchmark, only a 3.5% overhead is introduced. As indicated by the "Syscall Rate" column in Table 1, the measured slowdown directly corresponds to the rate at which processes execute traced system calls. Since the Latex benchmark executes only 16 traced system calls per second, it suffers the smallest slowdown.

To address the overhead imposed by the `proc` and related tracing facilities, we could implement TREC functionality in the kernel. Various tools such as Watchdogs [Bershad & Pinkerton 1988], Interposition Agents [Jones 1993], or SLIC [Ghormley et al. 1996] can be used to trace system call activity with little or no overhead. However, such tools often require root access to install, can be difficult to use without kernel source, and can also be difficult to distribute since kernel copyright restrictions may prevent distribution of source code. We opted for the user-level approach for portability, ease of distribution and installation. If performance becomes an issue, we believe switching to a kernel implementation will be straight-forward.

2.3 Limitations

As motivated earlier, a number of applications are able to benefit from TREC functionality. However, TREC can produce incorrect results for applications that base their results on non-deterministic or difficult-to-trace input. Following are some example behaviors likely to interact poorly with TREC applications:

- Programs must be deterministic and repeatable. Each of the programs that contribute to the creation of a file must behave the same way each time it is run, assuming that its own inputs have not changed. A compiler invoked with a given set of options will generally produce the same object file, as long as the source code has not changed. A simple example of a program that violates this restriction is UNIX `date`, whose output is virtually never the same twice. Any file that relies on the output of `date` cannot be guaranteed to be up-to-date, nor can it be reliably re-created. As discussed in Section 2.3, TREC does not currently automatically determine the class of applications that produce non-deterministic output.

- Programs cannot rely on user input. Related to the requirement for determinism, programs that rely on user input or GUI operations are not automatically re-creatable. For example, if an output file incorporates user-input to a text editor, TREC cannot accurately model program dependencies.

- Interaction with environment variables must be reproducible. TREC stores all environment variables in effect when a program runs. On subsequent runs of the same program, TREC can check to see if the currently set environment variables match the environment variables in effect when the program was originally run. If the variables do not match, the

Operation	Baseline	Traced	Syscall Rate	Added Overhead
open syscall	12.4 s	19.3 s	403 calls/s	54.8%
Compile	128.4 s	146.2 s	160 calls/s	13.9%
Latex	35.1 s	36.3 s	16 calls/s	3.5%

Table 1: This table describes the overhead introduced by adding TREC profiling.

program should be re-run (as opposed to using the cached results). TREC assumes that a program's interaction with environment variables is predictable: all other things being equal, a program run multiple times with the same set of environment variables should produce the same results. For some applications, the value of certain environment variables will not generally affect program results. In the future, we plan to allow TREC users to specify the set of environment variables that are known not to affect program results (e.g., REMOTE_HOST for certain CGI programs). It may also be possible to automate this process by repeatedly running the same program with different environment variable values in effect.

- File contents must be static, as long as the modification time has not changed. While seemingly a trivial constraint, certain special files do not follow this convention. Virtually all of the files in `/dev` violate this restriction. For example, each time the tape drive `/dev/rmt0` is read, it appears to have different data, for example, because an operator has exchanged one tape for another.

- File contents must be changed locally. For example, an NFS mounted file might be modified at any of a number of machines, not all of which may be traced by TREC. Applications requiring callbacks on file modification rely on TREC's ability to intercept all file updates.

- In general, TREC cannot profile programs that rely on network communication to produce their output[1]. For example, an application that communicates with a remote server may receive different results for each run of the program.

- Programs must run to completion while being profiled. For example, a program that terminates prematurely because it received a signal may not have generated complete dependency information, leaving TREC in an inconsistent or incorrect state.

[1]Note that techniques from the fault tolerance community could potentially address this limitation [Alvisi & Marzullo 1996].

Despite the limitations outlined above, we will demonstrate that TREC remains a useful tool in a number of different contexts. Our current approach to handling output files produced by programs that fall into the above categories is to allow users to specify a set of program names whose output cannot be cached or re-created. TREC uses a configuration file containing a list of programs whose output is potentially uncacheable. If any of these programs are involved in producing an output file, TREC caching is disabled (i.e., by transparent make or dynamic object caching). In the future, it should be possible to partially automate the process of determining the list of programs displaying "dangerous" behavior. For example, Solaris programs opening `/dev/tcp` can be assumed to be carrying out network communication.

Note that while TREC may be unable to transparently cache the results of certain programs, applications such as unmake can still provide valuable information to users about the origins of even uncacheable files. For example, if an image file is created using an interactive visual analysis tool, the output cannot be transparently re-created since user input was used to drive the result. However, unmake can still be used to identify the command executed to start the analysis tool, to determine the time the command was executed, or to enumerate all input files used during the creation of the image file.

3 Applications

In this section, we describe three utilities built on top of the TREC tracing module: unmake, transparent make, and dynamic web object caching.

3.1 Unmake

The TREC tracing module builds a process lineage tree as described in the previous section. To demonstrate the use and utility of this information, we describe a simple TREC service, unmake, that allows for a number of simple queries. For example, users might request infor-

mation about all processes, and their parents, that read a particular file.

As a concrete example, consider the shell script used to compile a simple program in Figure 2(a). When this script is run in a shell traced by TREC, the tracing module automatically builds lineage information for the processes executed by the script. The complete process lineage tree for producing the `test` executable is presented in Figure 2(b). Arrows between rectangles indicate process parent information. For the leaf processes, we indicate the files (indicated as ovals) read and written by each process. Notice that while the compilation script makes no reference to a header file (`test.h`) or to the assembler and loader programs, TREC is able to build full lineage information by observing the execution of all processes and sub-processes spawned by the compilation (e.g., TREC is able to transparently deduce a dependency to `test.h` by observing the fact that cpp read `test.h` for input).

The unmake module can be interactively queried for process lineage. Figure 3 shows partial output of a run for a query requesting lineage information for processes reading `test.c`. Unmake searches the lineage information for records where the set of input files contains `test.c`, in this case returning execution of the C pre-processor, `cpp`. Unmake returns the following information about the execution environment of all traced processes. All command line arguments, in addition to the environment variables, are listed (note that for brevity, the list of environment variables is truncated). All of the program's input and output files are listed along with a unique program ID (currently the UNIX pid) and the ID of the process's parent. The children field specifies all spawned processes (no processes were forked in the case of `cpp`). The query also recursively provides information on the process's parent, `gcc` in this case. While omitted from Figure 3, information on `/bin/sh`, the process which executed the compilation shell script, and `/bin/tcsh`, the root process of the TREC trace, is also returned.

While not currently implemented, unmake combined with a source control system or, more generally, a file system capable of transparently producing older file versions [Heydon et al. 1997] can be used to rollback to earlier versions of output files. For example, users debugging a program executable may use an interactive process lineage visualization tool, similar to the display in Figure 2(b), to identify input files that may have potentially introduced bugs. The user could then roll back to an earlier version of the suspect input file (using an appropriate file system or a source control system), instructing unmake to rebuild the output file with the new set of input files.

3.2 Transparent Make

The process and file lineage information produced by TREC can be used to build a more dynamic version of the traditional `make` utility, called transparent make (tmake). With traditional `make`, users are forced to learn a new language for specifying dependencies between input, intermediate, and output files, a process that can become cumbersome and error-prone for large development efforts. In contrast, tmake allows users to describe the process for creating output files more naturally; shell scripts (Figure 2(a) provides a simple example) describe the sequence of steps used to create an output file.

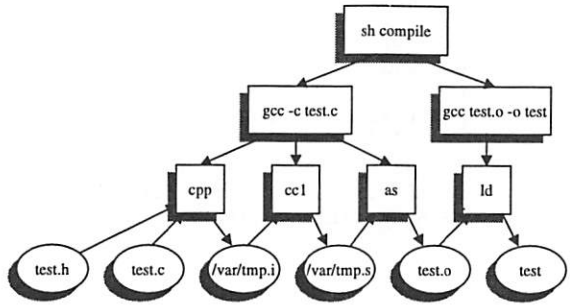
Thus, rather than forcing users to manually specify dependencies through Makefiles, TREC is able to dynamically determine dependencies by observing the execution of shell scripts. This approach has the following advantages: (i) eliminating user errors that may occur in specifying dependency information, (ii) dynamically updating dependency information as it changes, and (iii) eliminating the need to learn the Makefile specification language, which can be sometimes be complicated, restrictive, and/or error-prone. Of course, some users may be unable to create a shell script for the compilation of their program because they are accustomed to leveraging existing Makefile templates for compilation of their programs. In this case, tmake can bootstrap its execution by profiling the initial execution of a `make` command.

We currently have two different versions of tmake: a passive version that brings output up to date in response to a user command, and an active version that automatically updates output files whenever changes to an input file is detected. Both versions can be useful in different contexts. For example, active tmake may be appropriate when a summary file is produced based on a set of input files. All commands used to create the summary could be re-executed each time a data set changes while a visualization tool detects the changes to the file to update its display of the summary file. In this way, users could interactively manipulate data sets while visualizing the effects on a resulting graph. Passive tmake is likely more appropriate in compilation environments where users change multiple source files and wish to manually instruct tmake to re-synchronize output files.

The passive version of tmake provides an interface similar to traditional `make`. A shell script carries out tasks such as compilation while TREC builds process lineage

```
gcc -c test.c
gcc test.o -o test
```

(a) Compile Script



(b) Process Lineage

Figure 2: The top portion of the figure shows a simple shell script used to compile a program, `test`. *The bottom portion of the figure graphically depicts the process lineage tree produced by TREC-profiling of the shell script.*

information. However, passive tmake does not register callbacks on file write events. Rather, the user explicitly requests re-synchronization of the target of the shell script by re-executing the shell script. For each command in the shell script, tmake looks up the set of input files for the command and checks the last modified time of each input files. If any of the files have been modified since the last execution time of the command, the process is re-executed. Otherwise, the command is skipped. Also consider the case where the compile shell script is modified—for example, to add a new `-O` parameter to the compiler. In this case, tmake will re-execute any programs with modified command line parameters even if file dependencies have not changed since it will be unable to match the new process and command line parameters with any entries in its process lineage hierarchy. Thus, passive tmake functions similarly to `make`, while maintaining the advantages of implicitly determining dependency information and dynamically updating dependencies as they change (without requiring user intervention).

With active tmake, the tmake module registers a callback with the TREC tracing module when any file is opened for writing. When the callback is invoked, tmake checks if the file acted as input to any of the traced processes. If so, tmake notifies the user of this update and prompts for re-synchronization of the output files. On a user synchronize command, tmake re-executes the program that took the modified file as input with the same command line parameters and environment variables as the program's initial execution. Once the program completes, tmake recursively checks for further dependencies: if the output files of the just executed program acted as input for any of the program's parents in the lineage tree, the ancestor is in turn re-executed. This process is repeated until an output file is produced that did not act as input to any ancestor in the lineage tree. Assuming that the originally profiled program completed, this recursion is guaranteed to terminate because the dependency tree is constructed without cycles.

To demonstrate the workings of active tmake, consider the process lineage example from Figure 2(b). When the file `test.h` is modified (e.g., through an editor), TREC invokes a callback to the tmake module informing it of the change. Tmake then searches for all processes that used `test.h` as input, finding the cpp process. After asking for confirmation from the user, tmake re-executes cpp, noting that it produced an output file, `/var/tmp.i`. The process is repeated recursively, where tmake notices that the modified file was read by the cc1 process. Processes are executed in this way until ld produces a current version of the `test` executable. Recursion ends at this point since no process took `test` as its input. Note that if the `test` program is run from a shell, it is not classified as input to the shell since the `exec` system call (as opposed to `read`) is used to run the program. In the general case, files are considered to depend on the programs executed to create them. Thus, if the C compiler (`/bin/cc`) is updated, all C files will be re-compiled. However, the recursive check does not consider `exec` calls as an input dependency because, for example, it is pointless to re-execute a shell used to invoke an updated program.

Instead of prompting the user for permission to re-synchronize output files, tmake can be configured to

Query: read test.c

```
Parent ID: 28426 Program ID: 28428
Argv:   /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/cpp
        -lang-c -undef -D__GNUC__=2 -D__GNUC_MINOR__=7 -Dsun -Dsparc
        -Dunix -D__svr4__ -D__SVR4 -D__GCC_NEW_VARARGS__ -D__sun__
        -D__sparc__ -D__unix__ -D__svr4__ -D__SVR4 -D__GCC_NEW_VARARGS__
        -D__sun -D__sparc -D__unix -Asystem(unix) -Asystem(svr4)
        -Acpu(sparc) -Amachine(sparc) test.c /var/tmp/cca006u2.i
Envp:   COLLECT_GCC=gcc HOME=/homes/rivers/vahdat HOST=tolt HOSTNAME=tolt
        HOSTTYPE=sun4 LOGNAME=vahdat MACHTYPE=sparc OSTYPE=solaris
Children: (none)
Input:  test.c test.h /usr/include/sys/feature_tests.h /usr/lib/libc.so.1
        /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/include/stdio.h
        /usr/lib/libdl.so.1 /usr/platform/SUNW,Ultra-1/lib/libc_psr.so.1
Output: /var/tmp/cca006u2.i
========
Parent ID: 28424 Program ID: 28426
Argv:   gcc -c test.c
Envp:   HOME=/homes/rivers/vahdat HOST=tolt HOSTNAME=tolt HOSTTYPE=sun4
        LOGNAME=vahdat MACHTYPE=sparc OSTYPE=solaris
Children: 28428 28430 28432
Input:  /usr/lib/libc.so.1 /usr/lib/libdl.so.1
        /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/specs
        /usr/platform/SUNW,Ultra-1/lib/libc_psr.so.1
Output: (none)
```

Figure 3: This figure describes the results of a sample query, tracing back the lineage of the process that read the file `test.c`.

skip the prompt and to automatically re-creat output files when any input file is modified. However, such automatic re-synchronization can produce undefined behavior in the general case (e.g., users saving intermediate versions of program source files that will not compile). Of course, earlier work in optimistic make [Bubenik & Zwaenepoel 1989] has demonstrated the value of creating output files in anticipation of user requests. Thus, optimistic versions of output files could be created in temporary directories; once the user requests an update, a new version of the output file can be moved in place of the old one instead of waiting for the file to be re-created.

3.3 Dynamic Web Caching

In this subsection, we describe a third TREC example, dynamic web caching. This service is quite different in motivation and implementation from both the previous services, unmake and tmake. We begin by motivating the need for dynamic web caching and go on to describe how we modified an HTTP server to interact with TREC in order to provide this service.

3.3.1 Motivation

In response to the exponential growth of packets across the Internet, several researchers have proposed a number of caching schemes both to reduce the load on Internet backbones and to improve user response times [Gwertzman & Seltzer 1996, Chankhunthod et al. 1996, Zhang et al. 1997]. One early study [Danzig et al. 1993] found that strategically-placed caches could reduce FTP file traffic by as much as 50%. Similar studies of WWW traffic yielded similar results [Braun & Claffy 1994, Duska et al. 1997, Gribble & Brewer 1997].

We observe that any caching scheme will be limited by the fraction of web pages that are dynamically generated, and hence classified as uncacheable. For example, a CGI-bin program might be run to produce HTML in response to a user query (e.g., what are the show times at a movie theater) or to embed a different advertisement in the same logical page based on the identity of the requester. Approximately 20% of the queries to IBM's Web server for the 1996 Olympic games resulted in the dynamic generation of HTML (e.g., to get current medal standings) [Iyenger & Challenger 1997]. In general, the contents of such pages cannot be cached because the result of the program can change from execution to execution. Caching dynamic objects can be even more important for overall performance for the following rea-

sons: (i) An increasing percentage of web objects are being dynamically generated, (ii) a program must be run on the server side to generate such objects, increasing server load and (iii) client latency is generally limited by the minimum time required to run the program.

Our approach to reducing the overhead of busy web servers is to cache dynamically generated pages, using TREC to manage invalidations. Limitations to the type of dynamic objects that can be cached—for example, those that access a database—are described in Section 3.3.4. In this scheme, cache objects are stored in the file system under the name of the program used to generate them concatenated with any arguments to the program. Thus, a request for the object `http://www/cgi-bin/query?argument` might be cached locally in, for example, a file `/usr/local/apache/cache/cgi-bin/query?argument`. Subsequent accesses to the same CGI program with the same argument list can be returned from the disk cache, eliminating the need to `fork` and `exec` operations, and saving any computation time associated with the requested program. For example, consider user queries to a web site providing movie show times. Caching is attractive in this context because locality is likely present in the access pattern (popular movie at popular theater) and because the query results remain valid for an extended period of time (e.g., one week).

Of course, one problem with caching dynamic objects is maintaining cache consistency. The dynamically generated web objects often depend on a set of input files. For example, a consumer web site might provide an interface for users to interactively query for the latest pricing and availability information. Dynamic object caching can reduce server load by caching the replies to frequently made requests. However, all cached copies must be invalidated when pricing or availability information changes. As another example, a news site may dynamically generate a "front page" containing headlines and synopsis of news stories. Caching is also useful in this context since the same object will be delivered to all users for a certain time period. Once again, however, cached copies must be invalidated when the list of available stories is updated.

To address this need for invalidation, we use TREC to profile the execution of programs creating dynamic web objects. When TREC detects that an input file contributing to the creation of a cached object has been modified, one of two courses of action can be followed: (i) the file containing the cached copy of the web object is removed, forcing the web server to re-create it on the next user access, or (ii) the program which originally created the cached object can be re-executed to bring the cache up to date. Determining which approach is taken depends on the popularity of the object in question, the current load of the web server, and the cost of recomputing the object.

3.3.2 Implementation

To investigate the utility of dynamic web caching as described above, we modified Apache's HTTP server (version 1.2.4) in the following way. When a CGI object is requested, the server first checks for a file whose name matches the CGI object name concatenated with any arguments. If the file exists, its contents are returned without spawning a new process to carry out the request. If not present, a process is spawned to produce the desired results. The program's output is written to a file in parallel with the response to the requester. Currently, CGI arguments are assumed to be transmitted on the command line, corresponding to an HTTP GET request. In the future, it should be straightforward to modify Apache to include arguments transmitted in the HTTP header, corresponding to HTTP POST requests. Thus, subsequent requests are able to use a cached copy of the CGI object. File locking is used to ensure that partially generated results are not returned to users. We were able to make these changes by modifying approximately 50 lines of C code from the Apache distribution.

To allow for invalidations, the execution of CGI programs is profiled by TREC. Similar to transparent make, the dynamic web caching module registers callbacks for all files that act as input for CGI programs, requesting notification when any of the target set of files are modified. When such a callback is received, all CGI objects (cached output files) which depended on the modified file are removed, forcing the server to regenerate the result on the next user access. Since this level of dependency checking cannot guarantee consistency for all CGI objects (further discussed in Section 3.3.4), we allow the server administrator to specify a set of CGI programs that cannot be cached through an Apache configuration file.

3.3.3 Performance

To quantify the baseline performance benefits of caching CGI-bin program results, we measured the performance of retrieving CGI results for both our modified Apache server and the original, unmodified version. The mea-

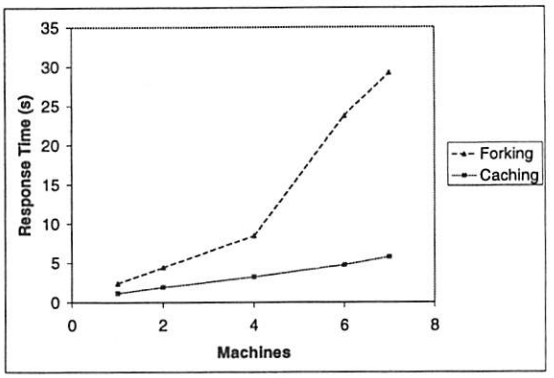


Figure 4: This figure describes the relative performance improvement introduced by CGI object caching.

surements were taken as follows. Eight Sun Ultra/1 workstations running Solaris 2.5.1 connected by a 10 Mb/s Ethernet switch were used for the experiments. One of the machines acted as the HTTP server running Apache 1.2.4. Between one and seven of the other machines acted as clients, continuously requesting the results of executing a small CGI script, printenv, which simply prints out the environment variables of the running CGI script. Each client machine forked 40 copies of the same script requesting 200 copies of the same script in a loop. Given the small size of the requests and the replies, network bandwidth was not the bottleneck.

Figure 4 describes the relative performance of the baseline vs. the modified Apache server under the above conditions. As a point of reference, the printenv script takes .05 seconds to run locally. One client requesting the CGI-script in a loop from unmodified Apache averages .18 seconds per request. Using the caching version of Apache, the CGI-script is retrieved in an average of .11 seconds, a 39% improvement over the baseline. From Figure 4, it is interesting to note that the relative performance of the caching CGI server improves with increased load. This improvement results from the high overhead of managing and context switching between address spaces as many CGI scripts are forked off by the baseline HTTP server under load.

We expect the above performance disparity to become even more pronounced as the CGI scripts become longer lived (recall that printenv exits after .05 seconds). To test this hypothesis, we re-ran our experiments with the baseline (uncaching) HTTP server returning the contents of a synthetic CGI program which takes 2.5 seconds to execute. Forty clients on one machine averaged 153 seconds to retrieve the object, while forty clients on each of two machines averaged 167 seconds to retrieve the object. Relative to the caching version of the server, the overhead of forking and executing the program slows the server down by nearly two orders of magnitude. Clearly, the savings from caching become more pronounced as the computation cost of the CGI object becomes more expensive.

3.3.4 Applicability

Our discussion of dynamic web object caching focuses on CGI programs. Given the inherent inefficiency of spawning a new process for dynamically generated content, a number of systems, such as ISAPI [Microsoft Corporation], NSAPI [Netscape], and FastCGI [Open Market], address this issue either by creating long-lived "server" processes responsible for creating dynamic content or by linking dynamic content producers into the servers address space. Relative to CGI scripts, these approaches offer better performance but sacrifice some of the simplicity of writing CGI scripts. However, TREC can still be used to improve the performance and functionality of these faster dynamic systems. For example, using TREC can eliminate the computation time associated with long-running scripts and any inter-process communication necessary to request dynamic generation of web objects. Thus, while TREC's baseline performance improvements would not be as impressive relative to systems that avoid fork and exec overhead, TREC still maintains the advantages of (i) using the familiar file system interface to cache content and (ii) eliminating any computation time required for dynamically generating web objects.

Another potential limitation of TREC for dynamic web object generation is the fact that many dynamic objects are generated as a result of queries to full-fledged databases. In essence, many web servers act as front ends to a sophisticated DBMS. For example, a user query for a price quote or item availability often translates into a database query. Since access to database relations cannot in general be modeled by simple file accesses (many databases are implemented on top of raw disks as opposed to the file system for example) , TREC cannot catch database updates, and hence cannot properly invalidate web objects based on out-of-date database values.

While the above limitation is inherent, we believe the performance improvements available from dynamic object caching argues for further research into active

databases. For example, research efforts into *materialized views* [Gupta et al. 1993, Gupta & Mumick 1995, Colby et al. 1996, Kawaguchi et al. 1996] in active databases [McCarthy & Dayal 1989, Stonebraker et al. 1990, Widom & Finkelstein 1990] has resulted in support for such views in many commercial database systems. Materialized views allow the results of a query to be updated as the tables (and individual cells) used to create the view are updated. Techniques similar to those employed by TREC are used to track view dependencies on individual cells and tables, and to set "triggers" to be fired when a table is modified so that any derived views can be updated as well. An interesting avenue of future research is to evaluate whether materialized database views can be used to cache the portion of dynamic web objects that generate database queries to obtain their results.

Another limitation faced by dynamic object caching is that, as described, TREC profiling and invalidation only allows for caching on the server-side. If such caching could be extended to Web proxies, performance could be further improved by caching dynamic objects closer to clients, potentially reducing both consumed wide-area bandwidth and user-perceived latency. One approach to addressing this limitation is to use a wide-area file system such as AFS [Howard et al. 1988] or WebFS [Vahdat et al. 1998] to store and to cache dynamic web objects as normal files. Thus, the wide-area file system can act as a shared file cache for both the HTTP server and interested proxies, with TREC invalidations maintaining relatively strong consistency semantics. Another approach is to allow proxy caches to cache dynamic objects with a TTL-based invalidation scheme [Chankhunthod et al. 1996, Gwertzman & Seltzer 1996, Squ 1996]. While this approach provides weaker consistency semantics, it is easier to deploy given the current Web infrastructure.

4 Related Work

Several systems have attempted to extend the automatic control of derived objects beyond the simple (but powerful) model used by `make`. DSEE [Leblang & Chase 1984, Leblang & McLean 1985], Odin [Clemm & Osterweil 1990] and Vesta [Levin & McJones 1993, Heydon et al. 1997] provide tools for modeling the behavior of programs, enabling the concise specification of derivation rules, and distributing changes to developers. Their declarative style suits large-scale programming environments, which are highly structured and employ a well-defined set of tools (compilers, linkers, etc.). None of these tools provide any assurance of correctness; as with `make`, the user is responsible for describing the complete set of dependencies relationships to the configuration manager. In contrast to unmake, users of these systems must tell the system how tools use files, whereas unmake simply observes and gathers the information in the background.

Odin relies heavily on the use of naming conventions: the name of a file fully specifies how it was derived. This restriction would not work well for the ad-hoc, highly parameterizable methodology used in less-structured environments. Like tmake, Odin implements transparent re-creation of files. A sentinel in Odin is a data object that is automatically regenerated (if necessary) at the time a user requests it, based on rules that were specified in advance for objects of its type.

VOV [RTDA], a configuration management toolkit, is similar to TREC, in that it observes program invocations to generate a trace of lineage information. However, VOV is limited to a specialized application domain (Electronic CAD), and it requires assistance from tool programmers. Each tool explicitly reports the files it will read and write. By contrast, unmake observes file-system activity at a low enough level that modifying tools to work with TREC is unnecessary.

Recently, a large body of research is being conducted in web caching. Harvest [Chankhunthod et al. 1996] and Squid [Squ 1996] are efforts into hierarchical web proxy caching. We believe that such caching efforts would benefit from our work in dynamic object caching. Gwertzman and Seltzer [Gwertzman & Seltzer 1996] recently proposed using the Alex protocol [Cate 1992] for maintaining cache consistency across the wide area. While this protocol provides weaker consistency guarantees than a wide-area file system, it would be simpler to deploy and could be used in our model for caching dynamic web objects at proxy caches. Finally, one proposal advocates using HTTP profiles to predict accesses to dynamically generated data, allowing servers to pregenerate potentially expensive pages in anticipation of user requests [Schechter et al. 1998].

Iyenger and Challenger [Iyenger & Challenger 1997] have implemented a caching system and API as part of IBM's web server that allows for caching of dynamic data. Their system allows for caching of dynamic data generated by arbitrary programs. Their work requires explicit invalidation of dynamically-generated content by the Web server, whereas TREC takes steps to automate this procedure. Further, TREC caches dynamic ob-

jects as normal files, simplifying system integration with existing Web servers.

The performance results presented in Section 2.2 here are similar to overhead studies of process migration and remote execution in Sprite [Douglis & Ousterhout 1991]. In Sprite, a number of system calls must be forwarded to the "home node" of a job for local processing. While the overhead of these operations in isolation is high, the overall perceived slowdown is tolerable because of the low frequency of forwarded operations. Similar to TREC, I/O system calls such as `read` and `write` are processed locally in Sprite.

5 Conclusions

The task of managing interactions between input and output files can be difficult. Further, the task of manually specifying such dependencies can be tedious and error-prone. We address this problem by introducing Transparent Result Caching (TREC), which automatically and transparently constructs dependency information by observing program behavior. To demonstrate its utility, we have described, built, and evaluated three sample TREC applications. Unmake allows users to query for process lineage information, returning the full chain of processes, command line parameters, and environment variables used to create a file. Transparent Make uses the process lineage information from Unmake to provide functionality similar to UNIX `make`, with the added advantage of freeing users from manually specifying file dependencies. Finally, Dynamic Web Object Caching allows web servers to coherently cache the results of dynamic web content such as CGI programs, with the potential of reducing server load and client latency.

Acknowledgments

Joel Fine did implementation work on an earlier version of transparent make/unmake running on Digital workstations and contributed to an earlier version of this paper. This work greatly benefited from discussions with Paul Eastham, Stefan Savage, Ashutosh Tiwary, and Geoff Voelker. In particular, comments from Fred Douglis, our shepherd, and the anonymous referees greatly improved both the content and presentation of this paper.

References

[Alvisi & Marzullo 1996] L. Alvisi and K. Marzullo. "Tradeoffs in Implementing Optimal Message Logging Protocols". In *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, June 1996.

[Apa 1995] *Apache HTTP Server Project*, 1995. `http://www.apache.org/`.

[Baker et al. 1991] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. "Measurements of a Distributed File System". In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 198–212, October 1991.

[Bershad & Pinkerton 1988] B. N. Bershad and C. B. Pinkerton. "Watchdogs—Extending the UNIX File System". *Computing Systems*, 1(2):169–188, Spring 1988.

[Braun & Claffy 1994] H.-W. Braun and K. Claffy. "Web Traffic Characterization: An Assessment of the Impact of Caching Documents From NCSA's Web Server". In *Second International World Wide Web Conference*, October 1994.

[Bubenik & Zwaenepoel 1989] R. Bubenik and W. Zwaenepoel. "Performance of Optimistic Make". In *Proceedings of Sigmetrics*, pp. 39–48, 1989.

[Cate 1992] V. Cate. "Alex – a Global Filesystem". In *Proceedings of the 1992 USENIX File System Workshop*, pp. 1–12, May 1992.

[Chankhunthod et al. 1996] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. "A Hierarchical Internet Object Cache". In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[Clemm & Osterweil 1990] G. Clemm and L. Osterweil. "A Mechanism for Environment Integration". *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, Jan 1990.

[Colby et al. 1996] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. "Algorithms for Derred View Maintenance". In *SIGMOD*, pp. 469–480, 1996.

[Danzig et al. 1993] P. B. Danzig, M. F. Schwartz, and R. S. Hall. "A Case for Caching File Objects Inside Internetworks". In *ACM SIGCOMM 93 Conference*, pp. 239–248, September 1993.

[Douglis & Ousterhout 1991] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software—Practice and Experience*, 21(8):757–85, August 1991.

[Dozier 1993] J. Dozier. Personal Communication, March 1993.

[Duska et al. 1997] B. Duska, D. Marwood, and M. J. Feeley. "The Measured Access Characteristics of World Wide Web Client Proxy Caches". In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[Ghormley et al. 1996] D. P. Ghormley, D. Petrou, and T. E. Anderson. "SLIC: Secure Loadable Interposition Code". Technical Report CSD-96-920, University of California at Berkeley, November 1996.

[Goldberg et al. 1996] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. "A Secure Environment for Untrusted Helper Applications". In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[Gribble & Brewer 1997] S. D. Gribble and E. A. Brewer. "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace". In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[Gupta & Mumick 1995] A. Gupta and I. S. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In *Data Engineering Bulletin*, June 1995.

[Gupta et al. 1993] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. "Maintaining View Incrementally". In *SIGMOD*, 1993.

[Gwertzman & Seltzer 1996] J. Gwertzman and M. Seltzer. "World-Wide Web Cache Consistency". In *Proceedings of the 1996 USENIX Technical Conference*, pp. 141–151, January 1996.

[Heydon et al. 1997] A. Heydon, J. Horning, R. Levin, T. Mann, and Y. Yu. "The Vesta-2 Software Description Language". Technical Report 1997-005, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1997.

[Howard et al. 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[Iyenger & Challenger 1997] A. Iyenger and J. Challenger. "Improving Web Server Performance by Caching Dynamic Data". In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, pp. 49–60, December 1997.

[Jones 1993] M. B. Jones. "Interposition Agents: Transparently Interposing User Code at the System Interface". In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 80–93, December 1993.

[Kawaguchi et al. 1996] A. Kawaguchi, D. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. "Concurrency Control Theory for Deferred Materialized Views". Unpublished, 1996.

[Leblang & Chase 1984] D. B. Leblang and R. P. Chase, Jr.. "Computer-Aided Software Engineering in a Distributed Workstation Environment". In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 104–112, May 1984.

[Leblang & McLean 1985] D. B. Leblang and G. D. McLean, Jr.. "Configuration management for large-scale software development efforts". In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 122–127, Harwichport, Massachusetts, June 1985.

[Levin & McJones 1993] R. Levin and P. R. McJones. "The Vesta Approach to Precise Configuration of Large Software Systems". Technical Report 105, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1993.

[McCarthy & Dayal 1989] D. R. McCarthy and U. Dayal. "The Architecure of an Active Data Base Management System". In *SIGMOD*, June 1989.

[Microsoft Corporation] "ISAPI Overview". `http://www.microsft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimr%g.htm`. Microsoft Corporation.

[Netscape] "The Server-Application Function and Netscape Server API". `http://www.netscape.com/newsref/srd/server_api.html`.

[Open Market] "Fastcgi". `http://www.fastcgi.com`.

[RTDA] "VOV". `http://www.rtda.com/vov.html`. Runtime Design Automation.

[Schechter et al. 1998] S. Schechter, M. Kirshnan, and M. D. Smith. "Using Path Profiles to Predict HTTP Requests". In *Proceedings of the Seventh International World Wide Web Conference*, April 1998.

[Squ 1996] *Squid Internet Object Cache*, 1996. `http://squid.nlanr.net/Squid/`.

[Stonebraker et al. 1990] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. "On Rules, Procedures, Caching and Views In Database Systems". In *SIGMOD*, May 1990.

[Vahdat et al. 1998] A. Vahdat, T. Anderson, and M. Dahlin. "WebOS: Operating System Services for Wide Area Applications". In *To appear in the Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[Widom & Finkelstein 1990] J. Widom and S. J. Finkelstein. "Set-Oriented Production Rules in Relational Database Systems". In *SIGMOD*, May 1990.

[Zhang et al. 1997] L. Zhang, S. Floyd, and V. Jacobsen. "Adaptive Web Caching". In *Web Caching Workshop*. National Laboratory for Applied Network Research, June 1997.

SLIC: An Extensibility System for Commodity Operating Systems

Douglas P. Ghormley
U.C. Berkeley
ghorm@cs.berkeley.edu

David Petrou
Carnegie Mellon University
dpetrou@cs.cmu.edu

Steven H. Rodrigues
Network Appliance, Inc.
steverod@netapp.com

Thomas E. Anderson
University of Washington
tom@cs.washington.edu

Abstract

Modern commodity operating systems are large and complex systems developed over many years by large teams of programmers, containing hundreds of thousands of lines of code. Consequently, it is extremely difficult to add significant new functionality to these systems. In response to this problem, a number of recent research projects have explored novel operating system architectures to support untrusted extensions, including SPIN, VINO, Exokernel, and Fluke. Unfortunately, these architectures require substantial implementation effort and are not generally available in commodity systems.

In contrast, by leveraging the technique of interposition, we have designed and implemented a prototype extension system called SLIC which requires only trivial operating system changes. SLIC efficiently inserts trusted extension code into commodity operating systems, enabling a large class of trusted extensions for existing commodity operating systems such as Solaris and Linux, while retaining full compatibility with existing application binaries. By interposing trusted extensions on existing kernel interfaces, our solution enables extensions which are protected from malicious applications, are enforced upon uncooperative applications, are composable with extensions from other third-party sources, and can be developed at the user-level using state-of-the-art development tools. We have used SLIC to implement and demonstrate a number of useful operating system extensions, including a patch to fix a security hole described in a CERT advisory, a simple encryption file system, and a restricted execution environment for arbitrary untrusted binaries. Performance measurements of the SLIC prototype demonstrate a one-time installation cost of 2-8 μsec and a per-extension invocation overhead commensurate with a procedure call.

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship.

1 Introduction

Modifying modern commodity operating systems is extremely difficult and costly. They are large, complex systems developed over many years by large teams of programmers and contain millions of lines of code. It is not unusual for major releases of commodity operating systems to be riddled with flaws introduced during development, typically requiring additional "bug fix" releases which may in turn introduce their own flaws. Compounding these problems, the development and debugging environments for operating system kernels are considerably behind the state of the art. Consequently, it is extremely difficult in practice to add significant new functionality to modern commodity operating systems [12, 1, 34].

Although modifying commodity operating systems is complex and difficult, the need to do so remains. There is a large catalog of operating system functionality which has not been widely deployed, in part because of the difficulty of modifying existing systems: load sharing [51], process migration [43, 12], fast communication primitives [6, 44], upcalls [9], distributed shared memory [25], user-level pagers [49], and novel schedulers [46, 13, 27]. In addition, security flaws are routinely discovered and reported by organizations such as Carnegie-Mellon's Computer Emergency Response Team (CERT) and the Department of Energy's Computer Incident Advisory Capability (CIAC). Despite the need for immediate repair to prevent wide exploitation of these flaws, the required patches can take weeks to become available [42].

This work aims to significantly simplify the process of evolving existing commodity operating systems by enabling new extensions which can manage global resources and/or enforce security guarantees. The ideal system which achieves this goal would possess a number of characteristics: it would require few or no modifications to existing operating systems or applications; it would introduce little overhead; multiple

extensions from independent, third-party sources could be active simultaneously; extensions would be protected from malicious applications and enforced upon uncooperative applications; and kernel extension developers would be able to make use of state-of-the-art user-level development and debugging tools.

Accomplishing this goal would enable independent software vendors (ISV's) to develop and deploy innovative operating system features. In particular, new operating system features developed by research projects could be transferred directly to end users without the need to convince or wait for operating system vendors to adopt the modifications. Furthermore, many CERT and CIAC security advisories normally require the system administrator to wait for a patch from the operating system vendor; instead, the advisory could directly include a small extension to correct the flaw, reducing the window of vulnerability dramatically.

Prior approaches to extending operating systems can be roughly divided into three categories: (i) re-engineering the operating system from the ground up, in the process making it easier to extend, (ii) incrementally re-engineering selected portions of the kernel, and (iii) adding extensions to existing systems without significant modification to either the operating system or its applications.

Over the years, a number of systems have attempted to reduce the cost of adding new operating system functionality by re-engineering the operating system to be extensible. Systems built using this approach include Hydra [48], SPIN [5], VINO [36], Exokernel [14], and Fluke [15]. While many of these systems have successfully demonstrated greatly reduced costs for adding new functionality, the initial cost of replacing existing commodity operating systems is prohibitive; for example, Microsoft spent over $300 M developing Windows NT [50]. Consequently, extensibility architectures developed using this approach will remain unavailable to the average user for the foreseeable future.

A small number of projects have taken the second approach of re-engineering certain kernel interfaces to reduce the complexity of adding new functionality at those interfaces. The `vnode` interface [23] is a prime example of this approach. However, applying this technique to make existing commodity operating systems generally extensible would require modifying and exposing all interfaces where additional functionality is desired, effectively re-engineering the majority of the operating system. Again, for the foreseeable future, such interfaces are unlikely to become generally available in commodity operating systems.

We take the third approach of adding functionality with only minor modifications to the underlying operating system and no modification to application code or binaries. Our work differs from earlier efforts in that our solution—kernel-level interposition of trusted extensions on kernel interfaces—is simple to implement, is efficient, requires no specialized hardware support, protects extensions from malicious or faulty applications, enforces extensions on uncooperative applications, and supports extension stacking. We believe that no other system provides this powerful combination of features for extending existing commodity operating systems.

Prior attempts to extend the operating system without significant modification suffered from significant limitations. Interposition Agents [20] leverages the Mach [1] system call redirection facility to transparently insert user-level extensions at the system call interface. However, because extensions run unprotected in the application's address space and require application cooperation, extensions cannot enforce security guarantees or manage shared resources for competing applications. Software Fault Isolation (SFI) [45] can be used to protect extensions from applications even when loaded in the same address space. Unfortunately, SFI requires a number of compiler optimizations to achieve low overhead and therefore cannot be applied efficiently to existing application binaries. Protected Shared Libraries [4] has the same capability as SFI without the need for compiler optimizations, but does not enforce extensions on applications.

To overcome the limitations of these systems, we have developed SLIC, a prototype system for efficiently inserting trusted extension code into existing operating systems with minor or no modifications to operating system source code. Conceptually, SLIC dynamically "hijacks" various kernel interfaces (such as the system call, signal, or virtual memory paging interfaces) and transparently reroutes events which cross that interface to extensions located either in the kernel (for performance) or at the user-level (for ease of development). Extensions both use and implement the intercepted kernel interface, enabling new functionality to be added to the system while the underlying kernel and existing application binaries remain oblivious to those extensions. SLIC dynamically interposes extensions on kernel interfaces by modifying jump tables or by binary patching kernel routines. The prototype currently runs on Solaris 2.5.1.

We have used the SLIC prototype to implement a number of extensions which would have been significantly more difficult to accomplish by other means. One extension patches a security flaw publicized by CERT [40]. A second extension encrypts file, while a third provides a restricted process execution environment.

The rest of this paper is organized as follows. Sec-

tion 2 provides background on interposition. In section 3, we describe the design, implementation, and performance of SLIC, our prototype interposition system. Three sample extensions and their performance are presented in section 4. In section 5 we discuss our experience with interposition as an extension tool, and the lessons we have learned about building system interfaces to support interposition effectively. Section 6 discusses related work while sections 7 and 8 close with future work and conclusions.

2 Interposition Background

Interposition is the process of capturing events crossing an interface boundary and forwarding those events to an interface *extension*. The extension performs some processing on the event and then either passes the event on to its original destination or forces the event to return. Figure 1 illustrates interposition on an interface by first one extension and then a second.

Interposition has a number of useful properties: it is *transparent*, *incremental*, and *composable*. Interposition is *transparent* because inserted extension code both uses and implements the original interface, enabling user applications and the kernel to remain oblivious to extension code.

Interposition is *incremental* since extensions need only capture the events that they are interested in. Extensions are not required to handle all events crossing the interposed interface boundary, enabling them to leverage the functionality of the existing interface; for example, an extension logging `fork()` system calls can use the underlying operating system's `write()` system call to store the log. Hence, extension writers only have to implement the desired extension functionality, not the functionality of the entire interface.

Because interposition maintains the original interface above and below the extension, it can be applied recursively, enabling multiple, independent extensions to *compose* their functionality. The right-hand side of Figure 1 shows the addition of a second extension to the extension stack. In the diagram, extensions A and B are oblivious of each other's presence, just as the application and kernel are oblivious to the presence of either extension.

The transparency, incrementality, and composability of interposition make it uniquely suitable for extending existing interfaces. Specifically, the transparency of interposition implies that interfaces not designed for extensibility can be extended. Incrementality simplifies extension development by ensuring that extensions need only provide the desired functionality without reimplementing substantial portions of the kernel. Composability means that multiple extensions provided by independent, third-party vendors can be applied to a single interface.

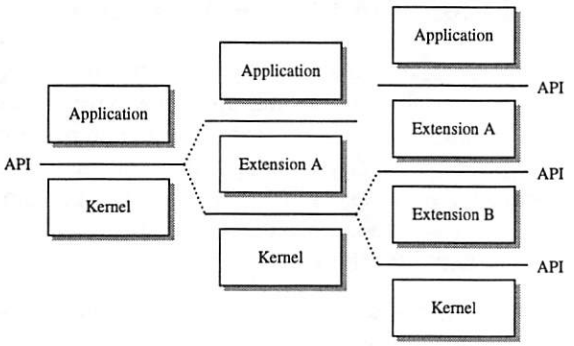


Figure 1: Interposing extensions on an interface. Solid lines indicate interfaces between components. Dotted lines illustrate the interposition of an extension on an interface. Events on the original interface are intercepted and routed through extension code. In this diagram, the original API on the left is maintained at each level on the right, making the interposed extension transparent to the applications, the kernel, and other extensions.

Using kernel-level interposition, extensions have a broad range of capabilities. Extensions can provide security guarantees (e.g., patching security flaws or providing access control lists), virtualize resources (providing cluster-wide process identifiers), modify data (transparently compressing or encrypting files), reroute events (sending events across the network for distributed systems extensions), or inspect events and data (tracing or logging).

However, interposition does have two important limitations. First, interposition requires a well-defined interface on which to capture events. Systems with poorly decomposed functionality may have few such interfaces. Second, new functionality can only be implemented in terms of existing functionality; for example, a cache-coherent file system can only be constructed through interposition if underlying layers expose a cache management mechanism in the file system interface [21].

Despite these limitations, the power and flexibility of interposition has led to its widespread use throughout modern computing systems. Forms of interposition can be found in virtual machines [19], object-oriented programming language systems [22], distributed file systems such as NFS [31], distributed shared memory systems such as TreadMarks [3], the 'pipe' construct of UNIX shells [30], World Wide Web proxy caches [7], MS-DOS terminate-and-stay-resident (TSR) utilities [32] and Macintosh toolbox extensions [11].

3 Design and Implementation

To investigate the suitability of interposition for adding new functionality to existing operating systems, we have designed and implemented SLIC, an interposition system for commodity Unix operating systems. SLIC leverages the transparency, incrementality, and composability characteristics of interposition to provide extensions with the following features:

Security: Extensions are protected from malicious or faulty applications and are enforced on uncooperative applications. This feature enables extensions which manage shared resources and/or enforce security guarantees. (Note that SLIC assumes that extensions are trusted. Other research efforts have addressed issues involved with untrusted extensions [45, 5, 18, 26, 29, 33].)

Ease of Development: During development and testing, extension writers are able to use state-of-the-art programming tools such as symbolic debuggers and performance analysis utilities.

Efficiency: Once development is complete, extensions impose minimal overhead on the system. Per-extension overhead is a few times the cost of a procedure call. Processes that don't use an extension experience a minimal performance slowdown.

3.1 SLIC Architecture

SLIC is comprised of multiple *dispatchers* and *extensions* as well as various *support routines*. Dispatchers are responsible for intercepting system events on a particular interface and for routing those events to interested extensions. Extensions receive events from one or more dispatchers and implement new operating system functionality. Support routines provide extensions with a simple, consistent interface to useful functionality such as memory allocation and synchronization primitives. Each dispatcher may provide additional support routines as appropriate for the interface; for example, our system call dispatcher provides routines to determine the children of a given process.

3.1.1 Dispatchers

Each SLIC dispatcher captures events on a single system interface. Dispatchers use two techniques to intercept interface events. For those interfaces which are invoked via jump tables, such as the system call, `vnode`, and virtual memory interfaces of Solaris, the dispatchers saves the original function address from the jump table and replaces it with the address of its own interception routine. For procedural interfaces which are called directly from various locations in the kernel, such as the signal interface, dispatchers must use binary patching to intercept events. The first few instructions of the relevant procedure are saved and replaced with instructions to jump to the dispatcher whenever the procedure is called. When the original routine needs to be invoked, the saved instructions are executed and control is returned to the original routine at the instruction following the binary patch. Using these techniques, SLIC dispatchers can capture interface invocations at the cost of a procedure call.

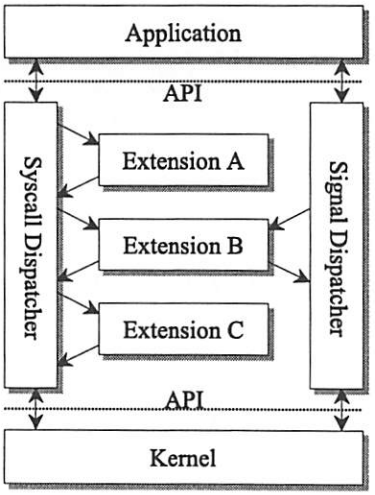


Figure 2: SLIC dispatchers and extensions. The dotted lines represent the interposed interface. Events crossing this interface are captured by a *dispatcher* and forwarded to one or more *extensions*.

Once an event has been captured by a dispatcher, that event is passed to interested extensions for processing. Figure 2 depicts the relationship between dispatchers and extensions. The dispatcher maintains an ordered list of extensions through which intercepted events flow down and return values flow back up. Dispatchers do not make any policy decisions regarding the ordering of extensions—extensions are ordered by the system administrator.

Figure 3 shows a simplified portion of the system call dispatcher interface. Extensions express interest in certain events using a small number of predicates defined by the dispatcher. For example, our system call dispatcher filters system call events based on process identifier and system call type; an extension that only wishes to trace the `open()` system call from process 4191 can specify this easily by invoking `Slic_TraceProc` for process 4191 and calling `Slic_RegisterHandler()` to register a handler only for the `open()` system call. Our signal dispatcher provides similar functionality, enabling filtering on pro-

```
struct Slic_SyscallInfo {
       int syscallNum;
       int args[];
};

struct Slic_ReturnInfo {
       bool forceReturn;
       int  returnValue;
       int  errno;
};

Slic_RegisterExtension(int dispatcherId);
Slic_RegisterHandler(int syscallNum,
        void (handler)(Slic_SyscallInfo *sysInfo,
                       Slic_ReturnInfo *retInfo));
Slic_TraceProc(int pid, bool traceAllChildren);
Slic_UntraceProc(int pid, bool untraceAllChildren);

Slic_IssueSyscall(Slic_SyscallInfo *syscallInfo,
                  Slic_ReturnInfo *returnInfo);
```

Figure 3: A simplified version of the interface exported by the system call dispatcher to extensions.

cess identifier and signal type.

Upon receiving an event, an extension has a number of options available: the extension can pass the unmodified event down the stack by simply calling `Slic_IssueSyscall()`, the extension can modify the event parameters in `struct Slic_SyscallInfo` and then pass it along, or by setting fields in the `Slic_ReturnInfo` structure, the extension can force the event to return back up the extension stack with an arbitrary return value or error condition (e.g., when a system call would exploit a known security hole). When an event is forced to return, extensions further down the call chain (including the original kernel routines) never see the event. The return value flows back up the chain in reverse order, allowing interested extensions to inspect or modify the returned value. Additionally, while processing one event, extensions may arbitrarily initiate other events on the same interface using `Slic_IssueSyscall()` with a different `Slic_SyscallInfo` structure, capturing the return values as needed. For example, an extension logging `fork()` system calls can initiate a `write()` system call to store the log on disk. Additional events generated by extensions are passed by the dispatcher to the next extension in the chain. To later extensions in the chain, including the kernel, these extension-initiated events are indistinguishable from application-initiated events and are executed with the privileges (and limitations) of the user process.

3.1.2 Extensions

The SLIC architecture enables extensions to be structured in two ways, supporting a tradeoff between ease

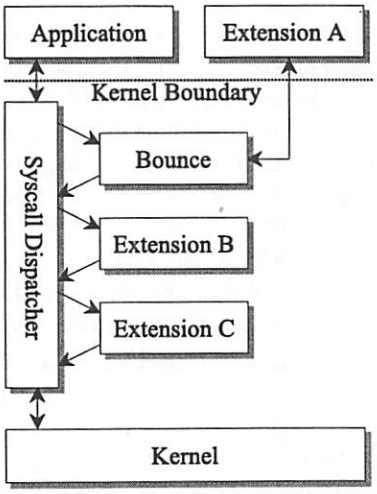


Figure 4: User-level extension environment. Extension A is being developed at the user level while extensions B and C are loaded in the kernel.

of extension development and performance. Extensions can be loaded at the user level or in the kernel, as shown in Figure 4, or as a combination of these types. The "Bounce" extension receives events from the dispatcher and hands them off to the user level extension; it also acts as an in-kernel proxy for the extension in order to access the dispatcher's utility functions from the user level. By providing extensions with the same interface whether at the user level or in the kernel, extensions can be easily developed and tested at the user level and then inserted into the kernel for performance. In all cases, the extensions are protected from and enforced upon uncoopertive applications.

Extensions loaded at the user level are each encapsulated in a separate user-level process. This architecture enables extension development to proceed just like standard user applications, with access to user-level libraries (e.g., communication libraries) and state-of-the-art development tools (e.g., symbolic debuggers and performance analysis utilities). User-level extensions are protected from malicious or faulty applications by virtue of running in a separate address space and are enforced on applications by the dispatcher which remains in the kernel. The disadvantage of this approach is that invoking the extension from the dispatcher requires costly context switches and kernel boundary crossings. This organization is similar to that employed by micro-kernels such as Mach [1] and `/proc` based systems such as Ufo [2] but differs in that it supports extension stacking.

To maximize performance once development and testing are complete, extensions can be loaded directly into the kernel where they are invoked directly from

the dispatcher with a procedure call. When events are frequent, this organization has considerably better performance than the user-level approach. In-kernel extensions are protected from malicious applications by virtue of being loaded in the protected kernel region of the address space; they are enforced on uncoopertive applications by the dispatcher. There are two limitations to this approach: (i) the kernel is not protected from malicious or faulty extension code and (ii) there is no support for user-level development tools.

Using a simple upcall/downcall interface, SLIC extensions can also use both models simultaneously. Performance-critical sections of an extension can be located in the kernel, while functionality that is rarely used or which requires access to user-level libraries can be located in a user-level server [37].

3.2 SLIC Implementation

The current implementation of SLIC provides dispatchers on the system call and signal interfaces of Solaris 2.5.1 running on SPARC 10, 20, and UltraSPARC workstations[1]. To minimize kernel modifications, all SLIC components—dispatchers, extensions, and support routines—are dynamically loaded into the kernel as loadable device drivers.

Solaris system calls are routed through the `sysent` table, which contains function pointers to the appropriate system call routines. The system call dispatcher intercepts system call events by replacing entries in this table with pointers to its own dispatch function. Solaris signal delivery proceeds through the `sigaddq()`, `sigaddqa()`, and `sigtoproc()` functions. The signal dispatcher intercepts signals by binary patching these functions at run-time. To enable binary patching, we changed one line of Solaris source code to make the kernel code writable by the SLIC dispatchers, although it should be possible to accomplish this without source code changes by manipulating the memory management hardware directly when SLIC is installed.

The current implementation catches events within the kernel, rather than at the machine level. System calls, for example, are caught at the `sysent` table rather than upon execution of the `trap` instruction. While the two approaches are conceptually similar, intercepting events within the kernel leverages a significant amount of machine-specific code which considerably simplifies the prototype.

[1] Although the SLIC prototype is implemented on Solaris, the principles underlying SLIC are generally applicable; we believe SLIC could easily be ported to other UNIX operating systems.

3.2.1 Shadow Structures

SLIC dispatchers and extensions need a way to record state that persists across event invocations. For instance, the system call dispatcher needs to keep track of which processes are marked for tracing. The appropriate place to store this information is in the process table or the thread structures, but in many operating systems, the size and organization of these structures are compiled into system utilities and other kernel modules.

To minimize modifications to the operating system, we implement *shadow structures* for processes and threads to store state for SLIC dispatchers and extensions. A naïve approach to implementing shadow structures would be to modify process creation and cleanup routines to include a call to manage the shadow structures. Instead, SLIC uses interposition to maintain shadow structures without kernel modifications. Shadow structures are created on demand, when a dispatcher or extension attempts to access a shadow structure. Initialization routines allocate space for the shadow structure and initialize it with a pointer to the underlying kernel's structures. To remove shadow process structures, SLIC interposes on the `wait()` and `waitid()` system calls to detect process death. In Unix, all processes must be waited on, even if by the `init` process. By observing the return values to the `wait()` and `waitid()` system calls, SLIC determines which processes have exited and removes the corresponding shadow structure. Similarly, thread shadow structures can be deleted either on a `thr_exit()` system call or when the thread's process exits.

3.2.2 System Call Buffers

System calls transfer two forms of data: pass-by-value arguments and pass-by-address memory buffers. Inspecting or modifying the pass-by-value arguments in an extension is straightforward. Inspecting or modifying memory buffers located in an application's address space, however, requires more care.

There is a window of vulnerability between the time when an extension inspects or modifies a user-level data buffer and the time when the underlying operating system executes the system call. Other threads in the application can exploit this window of vulnerability to alter the buffer, effectively circumventing any security checks performed by an extension, thus violating the requirement that extensions be enforced on applications. For example, an application may be able to circumvent an access control list extension of the file system by changing the path name of the `open()` system call during this window of vulnerability.

To prevent this situation, before a memory buffer can be inspected or modified by an extension, that buffer

must be protected from modification by the application. Copying these buffers to the kernel provides the necessary protection but introduces a different problem. During system call execution, Unix kernels perform protection checks on memory buffer accesses, requiring that those buffers be located in the application's address space. These checks, performed in the kernel's `copyin()` and `copyout()` routines, are normally necessary to prevent malicious applications from accessing sensitive kernel data. Once a buffer has been copied into the kernel by an extension, the kernel's own security checks will fail when the underlying kernel routines attempt to copy the data, disallowing access to the buffer.

Our solution to this problem is to again apply interposition. SLIC maintains a per-thread list of valid, in-kernel extension buffers and interposes on the `copyin()` and `copyout()` routines in order to permit access to these buffers when appropriate. When an extension allocates an in-kernel buffer to a thread, SLIC adds the address and length of the allocated region to that thread's valid list. When the `copyin()` or `copyout()` routines are invoked, the interposed code checks the target address against the thread's valid list. If the address is a valid extension buffer, then instead of the original `copyin()`/`copyout()` routines, the kernel's `bcopy()` routine is invoked to copy the data to its final location. If the address is not a valid extension buffer, then control is passed to the original copy routine which performs the normal protection checks.

Note that this process introduces an extra copy for data buffers; buffers are copied into the kernel for use by extensions and are later copied again by the underlying kernel routines. Eliminating the extra copy is impractical since numerous kernel routines expect buffers to be copied to stack frames which have not yet been allocated when the extension is invoked.

3.3 Microbenchmarks

To measure the overhead imposed by SLIC on system calls and signals, we ran three microbenchmarks on a 167MHz UltraSPARC running Solaris 2.5.1. Unless otherwise noted, benchmark timings are averaged over 100,000 runs. The first microbenchmark performs a `getpid()` system call, which in Solaris is essentially a null system call. This microbenchmark measures the raw overhead of the system call dispatcher, but does not invoke the modified copy routines since there are no buffers involved. To quantify the overhead resulting from our interposition on the copy routines, the second microbenchmark performs a `sigprocmask()` system call which involves a memory copy of 16 bytes. The `kill()` microbenchmark measures the overhead of the signal dispatcher. It involves a single process sending itself a `SIGUSR1` signal. We use a single process to avoid context switches, thus maximizing the effect of our overhead. Because of the longer run time, results for this benchmark are averaged over 10,000 runs.

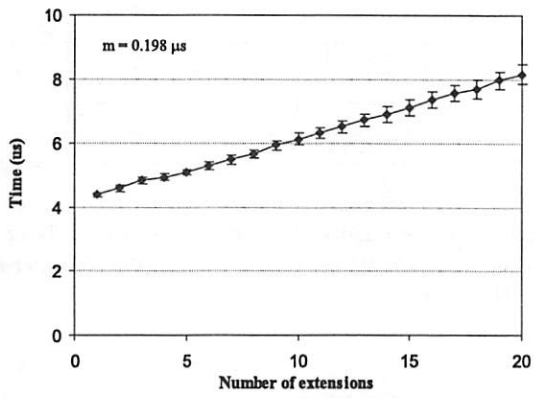


Figure 5: `getpid()` microbenchmark performance with a varying number of extensions. The time on an unmodified system is 2.82 µsec. Error bars indicate one standard deviation above and below the average.

We tested our microbenchmarks with various configurations of SLIC: an unmodified system, a system with SLIC dispatchers but no extensions installed, and a system with the SLIC dispatchers and multiple extensions installed. The results of the `getpid()` benchmark with a varying number of extensions are presented in Figure 5. The incremental cost of adding a new extension is statistically similar for all three benchmarks. Table 1 presents the one-time overhead cost of interposition for each of these benchmarks as well as the time to invoke a user-level extension.

The base overhead of the SLIC dispatchers is approximately 1.5 µsec for system calls and 5 µsec for signals. The incremental cost of loading additional extensions is approximately 0.2 µsec for both dispatchers. The extensions used for these measurements are null extensions which inspect events arguments but not event return values. Inspecting the return values in an extension consumes two additional stack frames and SPARC register windows—one is consumed by the extension while awaiting the return value and a second is consumed by the dispatcher which invoked the extension. Consequently, inspecting return values increases the per-extension cost from 0.2 µsec in Figure 5 to 1.60 µsec. The average cost of a standard procedure call which spills a register window on this platform is approximately 0.8 µsec.

There are two reasons why the overhead for the signal dispatcher is higher than that of the system call dis-

	getpid()		sigprocmask()		kill()	
	Time (µs)	σ	Time (µs)	σ	Time (µs)	σ
Unmodified system	2.82	0.05	3.75	0.20	103.52	1.56
SLIC, no extensions	3.16	0.05	4.19	0.10	108.76	1.53
SLIC, 1 null extension	4.38	0.07	5.56	0.09	109.99	1.49
SLIC, user-level extension	61.83	0.81	60.20	0.88	187.67	2.00

Table 1: Microbenchmark performance of SLIC. For each benchmark, the table shows the average elapsed run time in microseconds and the standard deviation across the runs. The `getpid()` and `sigprocmask()` tests were run with the system call dispatcher loaded while the `kill()` test was run with the signal dispatcher loaded. User-level extension results are averaged over 10,000 runs.

patcher. First, the signal interface is more complicated than the system call interface. Whereas system call events have a calling thread, a system call number, and system call arguments, the signal interface has a calling thread, a signal number, a target process, and an optional target thread. The shadow structure for each process and thread must be determined before extension processing can proceed; there are consequently more shadow structure lookups for the signal interface than the system call interface. The second difference is that intercepting signal events sometimes requires calling thread to pay two interposition costs. Specifically, in Solaris, the `sigaddq()` and `sigaddqa()` routines perform signal queueing side-effects which SLIC extensions may need to leverage; both of these routines call `sigtoproc()` in order to do further signal processing. However, some routines in the kernel invoke `sigtoproc()` directly, requiring SLIC to interpose on that as well to ensure that all signals are seen by the extensions. Those code paths in the kernel which invoke `sigaddq()` or `sigaddqa()` consequently experience the interposition overhead twice.

4 Extensions

To demonstrate the functionality and performance of SLIC, we have implemented prototypes of a variety of extensions: a security patch for a recent CERT advisory, an encryption file system, and a restricted execution environment. Without SLIC, responding to the CERT advisory would have required disabling `admintool` while awaiting a patch, and the encryption file system and restricted execution environment extensions would have required substantial kernel source code modification to achieve the same functionality and performance. Further, without SLIC, adding these features to an existing kernel would require *ad hoc* changes rather than providing a general solution that can be leveraged for future extensions.

4.1 CERT Advisory Extension

The Computer Emergency Response Team (CERT) regularly provides the Internet community with information regarding system security problems. Whenever possible, these advisories include information on how to resolve the reported problem. However, due to the lack of extensibility in existing systems, frequently this advice is to completely disable the insecure feature [39, 40, 41]. However, using SLIC, many of these advisories could be accompanied by small extensions which would resolve the problems without requiring changes to kernel source code. Though operating system vendors do respond to these advisories by supplying patches, those patches can take weeks to become available [42].

To demonstrate patching a security hole in this manner, we have implemented an extension to patch a security hole discovered in the Solaris `admintool` [40] which allowed unprivileged users to truncate arbitrary files. The `admintool` utility creates a local lock file to control access to shared files. By creating a symbolic link at the lock file location, malicious users could cause arbitrary files to be truncated when the lock file was created. Our 100-line extension monitors file operations, preventing symbolic links from being created at the lock location. SLIC thus corrects the security problem while maintaining continued use of `admintool`.

4.2 Encryption File System

In a distributed file system, maintaining file privacy is a primary concern. In a networked environment with a central file server, traditional Unix file protections can be easily circumvented by monitoring network traffic. To protect sensitive files, users may use encryption tools such as PGP [16]. However, stand-alone encryption tools and libraries can be time consuming or cumbersome to use and are not easily integrated with existing applications. A more effective method of ensuring file security is to support file encryption directly in the file system, transparently encrypting file writes

and decrypting file reads when communicating with the server. Rather than rewriting all file systems to support encryption, an easier approach is to implement a single encryption extension which interposes on all file traffic.

We have implemented a simple extension to demonstrate the feasibility of file system encryption using SLIC. This extension implements a trivial *exclusive-or* encryption algorithm similar to that used to test VINO [33]. The prototype extension watches for `open()` and `creat()` system calls of files with a particular suffix and then records the process identifier and the file descriptor returned to the application. On subsequent `read()` or `write()` system calls to these file descriptors, the extension applies a byte-wise `xor` on the data. Key management and encryption algorithms are orthogonal to our demonstration that interposition provides a simple yet powerful means of transforming file system data.

4.3 Restricted Execution Environment

In UNIX, processes run by a user have access to all of the resources granted to that user. There are many cases, however, in which the user does not fully trust the program being run. For example, programs downloaded from untrusted sources may actually be Trojan horses designed to steal or destroy information [47, 10]. In addition, there are cases in which the user trusts the program, but not the data being processed, as in the case of web browser helper applications used by web browsers to display various data formats. Input data could potentially exploit bugs in helper applications such as `ghostview` to insert viruses into the system [38].

The tracing facility of the standard Solaris `/proc` file system is one method that has been used to construct a restricted execution environment [17]. Potentially insecure system calls are captured and then selectively denied or altered. The `/proc` approach, however, suffers from two primary shortcomings. First, intercepting system calls using `/proc` is expensive, requiring two context switches over the base system call overhead. This is especially problematic for system-call intensive applications. Second, systems using `/proc` cannot properly handle the system call buffer problem described in section 3.2.2. While `/proc` does enable an extension to inspect system call buffer data, a multi-threaded application may maliciously modify the buffer between extension validation of the buffer and kernel use of the buffer, effectively subverting the security system. Using SLIC we have implemented a restricted execution environment extension that does not have these limitations.

This extension is a modified version of the Janus system [17] and provides the user with a configurable security environment. For example, applications can be given a subset of read/write/execute access to any number of directory subtrees. The ability to `fork()` or to perform a variety of other system calls can be disabled. Any attempts by the application to perform a restricted operation results in the extension returning an `EPERM` error. The extension monitors only the subset of system calls necessary to maintain the security guarantees, thus minimizing overhead. When traced applications invoke restricted system calls, the extension checks the arguments to the call and determines if the call should be allowed or denied. The restricted environment used for benchmarking denies 45 system calls outright (e.g., `chown()`) and performs security checks, such as checking the path or file access permissions, for 21 additional system calls (e.g., `rmdir()`).

4.4 Performance

To evaluate the impact of these extensions on system performance, we ran the extensions under three benchmarks: the Modified Andrew Benchmark [28], a TEX compilation of a 494-page (1.32 MB) document, and a `gcc` compilation of `emacs-19.34` without support for X Windows. The Modified Andrew Benchmark consists of multiple phases which create directory subtrees, copy files, search file attributes via `find`, search files for a text string via `grep`, and compile files. While this benchmark fits entirely in the file cache of modern systems and is therefore no longer useful for measuring file system performance, this benchmark is useful for exposing the overhead imposed by SLIC. The TEX and `gcc` benchmarks were chosen to be representative of document processing and compilation workloads. All measurements were run on a 167MHz UltraSPARC running Solaris 2.5.1 with all benchmark data files placed in a memory-mounted `/tmp` file system. Table 2 reports some relevant statistics for each benchmark.

Table 3 presents the results of running the benchmarks with each extension as well as with all extensions simultaneously. Since the CERT extension does not catch any system calls issued by the benchmarks, as indicated by Table 2, it effectively acts as a null extension for these tests; consequently, the overhead depicted in Table 3 for the benchmarks running on CERT is exclusively due to the SLIC dispatchers and infrastructure. There are three anomalies in the table worthy of note: the MAB and `gcc` benchmarks appears to run faster with all three extensions loaded than with just the Encrypt extension loaded and the `gcc` benchmark appears to run faster with SLIC loaded than on the baseline system. However, an inspection of the standard de-

	Procs	Total System Calls	System Calls Caught					
			CERT		Encrypt		REE	
MAB	471	40500	0	0%	2732	7%	9246	23%
TEX	3	2748	0	0%	1635	60%	703	25%
gcc	379	140656	0	0%	11031	8%	47888	34%

Table 2: Benchmark characterization. For each benchmark, this table presents the total number of processes created during a run, the total number of system calls issued by those processes, and the number and percent of those system calls which are caught by each extension. "MAB" is the Modified Andrew Benchmark and "REE" is the Restricted Execution Environment.

	MAB			TEX			gcc		
	Time (*s*)	σ	S	Time (*s*)	σ	S	Time (*s*)	σ	S
Baseline	15.71	0.10		14.50	0.30		160.30	1.99	
SLIC, no extensions	15.92	0.16	1%	15.06	0.23	4%	159.13	0.34	-1%
CERT	16.12	0.30	3%	15.09	0.39	4%	160.11	0.55	0%
Encrypt	17.69	0.85	13%	15.87	0.53	9%	168.30	0.53	5%
REE	15.93	0.33	1%	15.15	0.73	4%	160.89	1.63	0%
CERT + Encrypt + REE	17.24	0.09	10%	15.90	0.37	10%	166.98	3.43	4%

Table 3: Benchmark performance on sample extensions. "Baseline" represents a machine without any SLIC dispatchers or extensions loaded. "SLIC, no extensions" represents dispatchers loaded, but no extensions; this row measures the effect of the SLIC interposition overhead. The rows labeled "CERT", "Encrypt" and "REE" present the benchmark elapsed times with a single extension loaded. The last line shows benchmark performance with all three extensions interposing simultaneously. For each benchmark, the table shows the average elapsed run time in seconds, the standard deviation across the runs, and the percent slowdown ("S").

viation in each of these cases reveals that the anomalies are well within one standard deviation of the mean and are likely to be experimental variance. Though SLIC imposes a certain amount of overhead on applications, the last line in Table 3 illustrates that much of the overhead experienced by the benchmarks is due to the SLIC dispatcher infrastructure, a cost which is only paid once; the per-extension overhead is small for these workloads.

5 Interposition Evaluation

This section describes our experiences with implementing SLIC and presents a number of general principles for developing interfaces that are conducive to interposition. Although SLIC was designed for and will work with existing operating systems, there are a number of improvements that can be made to make these systems more interposition-friendly. We have drawn these lessons from our implementations of the system call and signal dispatchers for Solaris 2.5.1, as well as preliminary analyses of the virtual memory mechanism of Solaris, the process scheduler interface of FreeBSD 2.2.5R, and the system call interface of Linux 2.0.

The problems that we have encountered can be divided into four categories: the asymmetric trust mechanisms of the system call interface, the implicit event information in the interfaces we interposed on, an incomplete decomposition of functionality in the system, and miscellaneous implementation issues.

5.1 Asymmetric Trust Mechanisms

The solution adopted by the current prototype to correctly handle system call buffers (see section 3.2.2) introduces a second copy for those data buffers which must be securely examined by an extension. This second copy is not fundamental to interposition but rather arises out of idiosyncrasies of the Solaris system call routines. In Solaris, as in most Unix systems, system call buffers must be copied into the kernel before they can be used. The copies are only necessary on interfaces such as the system call interface which have an asymmetry of trust—applications trust the kernel, but the kernel does not trust applications. Extensions which interpose on such interfaces are viewed by the kernel as being part of the untrusted user application. Interfaces which have a symmetry of trust do not need to copy data buffers before using them because it is assumed that no other threads in the trusted domain will maliciously modify the buffers to subvert the system.

Consequently, the ideal interface for interposition is one with a symmetry of trust. For the system call in-

terface, this would mean interposing on events *after* the data buffer addresses had been validated and the buffers themselves had been copied into the kernel. In Solaris, this is impractical because the copy routine invocations are scattered throughout the system call handler routines. The ideal interposition-friendly interface would perform any required security checks and copies in separate routines before invoking system call handlers. This approach would eliminate the unnecessary second copies and would remove the need for SLIC to interpose on the kernel's copy routines.

5.2 Implicit Event Information

Many interfaces in today's operating systems rely on *implicit information*—information that is not passed directly as an argument to the event but rather is stored in global data structures. For example, when a system call is made, the process identifier of the application is not passed to the system call handler. Similarly, the return value of a system call is not returned explicitly according to normal calling conventions, but is stored in the kernel's process structure. The credentials determining a process's right to open a file or send a signal are also stored in the process structure. Accessing this implicit information requires an understanding of complicated kernel structures and kernel locking conventions.

To minimize the dependency of extensions on a particular version of the operating system, SLIC dispatchers provide extensions with simple utility functions to access a variety of implicit event information, hiding the details of kernel data structures and locking conventions. Unfortunately, providing these functions in the dispatchers increases the dependency of those dispatchers on the particular version of the operating system. An interposition-friendly interface would make implicit event information readily accessible when an event is raised, enabling SLIC to be more easily ported to new operating system versions.

Implicit event information also limits the functionality of extensions, as in the case of implicit credentials. If extensions cannot modify event credentials, then extensions would be restricted to the rights of the calling thread. For example, an extension could not write to files owned by `root` unless invoked from a `root` process. A naïve approach to circumvent this problem would be to enable extensions to modify a process's credentials stored in global data structures for the duration of the intercepted event. However, in a multithreading environment, other threads running concurrently may access the modified credentials, producing unpredictable results. Including credentials as explicit parameters to the event enables extensions to modify the credentials as necessary.

The `ioctl()` system call poses a different problem for interposition. Originally designed as a way to manage arbitrary devices, `ioctl()` system calls have a variable number of parameters, nearly any of which may reference arbitrary memory buffers which may in turn contain pointers to other memory buffers. The meaning of the arguments and the structure of each buffer is defined by the particular device driver. Hence an extension cannot know in advance how to handle the arguments of any given `ioctl()` call. This is problematic for extensions such as system call logging or security extensions which may need to understand the arguments of an `ioctl()`. While it is possible to derive some information about device/application interactions by interposing on the `copyin()/copyout()` routines, general interpretation of `ioctl()` semantics is difficult. Events with run-time determined semantics are not conducive to interposition.

5.3 Separation of Policy and Mechanism

To reduce the dependencies on a particular hardware platform or operating system version, extensions should interpose on system policies while leveraging system mechanisms. A clean separation of policy and mechanism in the compiled kernel is therefore critical. However, this principle of decomposition is violated by today's operating systems. While experimenting with extending the scheduler interface in FreeBSD 2.2.5R, we found that a single scheduler routine, `cpu_switch()`, implements both the policy of selecting the next process to run as well as the actual context switch mechanism. Enabling interposition on the policy while leveraging the existing mechanism required separating the policy and mechanism into two routines, creating a procedural interface which could be interposed on.

A variation of this lack of separation occurs with the Linux routines for accessing system call data buffers, `getuser()` and `putuser()`. These routines are inlined at compile time, making interposition at run time extremely difficult. Interposing on the copy routines requires modifying the Linux source to disable inlining of those copy routines, again creating a procedural interface that can be interposed on, albeit at a small cost in performance.

5.4 Miscellaneous Issues

Our method of using binary patching to intercept procedural invocations is simple to implement, but requires that kernel code be writable. Unfortunately, Solaris is loaded such that the kernel code is read-only,

preventing binary patching. By modifying a single line of Solaris source, we were able to make the kernel code writable.

Managing shadow structures for processes and threads complicates SLIC and increases kernel memory consumption. Adding a hook to the kernel's process and thread structures would eliminate these problems.

6 Related Work

There has been a considerable amount of recent work [36, 5, 14, 15] that has explored novel kernel designs for extensible operating systems. Of these systems, SPIN [5] and VINO [36] are the closest in concept to our work. Both offer extensibility through interposition on a number of kernel interfaces, but have explicitly crafted those interfaces for extensibility. SPIN and VINO also aggressively focus on ensuring kernel protection from untrusted extensions, SPIN by using a type-safe language [35, 18], and VINO through software fault isolation [45] and in-kernel transactions [33]. In contrast, SLIC assumes trusted extensions and focuses on an evaluation of the technique of interposition and its suitability for legacy operating systems.

Interposition Agents [20] demonstrated the usefulness of constructing extensions in terms of the high-level abstractions of an interface (such as path names), rather than the low-level events crossing that interface (such as `open()`). Interposition Agents used the system call redirection facility of Mach which bounces system calls to extensions linked into an application's address space. Consequently, extensions are neither protected from nor enforced on applications and thus cannot implement security extensions or share data between distrustful applications. Additionally, the multiple protection boundary crossings limit the performance of the system. SLIC enables high-performance interposition that is both enforced on and protected from applications, enabling a significantly larger class of extensions. In principle, the toolkit presented in [20] could be ported to SLIC, further simplifying the process of extension development.

COLA [24] enables interposition at the system call interface, but without any modification of the operating system kernel. It operates through interposition at the library level and consequently suffers from the same security drawbacks as the Mach interposition technology described above.

Protected Shared Libraries (PSL) [4] enables extensions to be securely loaded into an application's address space, so that user programs cannot access or modify extension code or data. PSL does not provide a mechanism for enforcing extensions on applications, which SLIC does. PSL is primarily intended for adding new interfaces to a system, although it could be combined with the interposition mechanisms used in SLIC to enable modification of existing interfaces. Finally, the PSL protection technique relies on per-thread segment protections supported by the IBM RS/6000 architecture, while the principles in SLIC are generally applicable across a variety of operating system platforms.

Disco [8] and Fluke [15] are virtual machine monitors which use strategies similar to those of SLIC for different purposes. Disco uses interposition and binary rewriting to ease the implementation of operating systems for new architectures, rather than adding extensibility to existing operating systems, as SLIC does. Fluke uses interposition for extensibility, relying on a heavy decomposition of services into nested process domains, instead of adding extensibility to an existing kernel.

7 Future Work

While interposition enables extension stacking, the mechanism cannot guarantee that the extensions will be compatible with each other. Some extension combinations may only impair performance (e.g., for many types of data, it is more efficient to execute a data compression extension before a data encryption extension) while others may impair correctness (e.g., an extension which needs to inspect the magic number at the head of executables may run incorrectly when invoked after an encryption extension). Prior experiences with conflicts among MS-DOS terminate-and-stay-resident (TSR) utilities and Macintosh toolkit extensions indicate that a method for identifying and managing conflicts among incompatible extensions is sorely needed.

8 Conclusions

This paper has examined the utility of interposition as a mechanism for making commodity operating systems extensible. We have shown that interposition is suitable to a number of useful extensions, and we have presented a prototype system, SLIC, which enables operating system extensibility through interposition in Solaris with minimal kernel source modifications. SLIC demonstrates that extending an existing operating system can be done in a manner that is protected from applications, enforced upon uncooperative applications, and efficient, while combining the development and testing advantages of user-level extensions with the performance of kernel extensions.

We have also examined the problems found in transparently extending operating system functionality, such as the asymmetric trust of the system call interface and implicit event information. Drawing from experiences with these problems, we presented a number of lessons that can be used by operating systems designers to provide interfaces which are conducive to interposition. Foremost is the imperative to maintain clear procedural barriers between operating system policy and mechanism. Additionally, to reduce the effort necessary in implementing an interposition mechanism, interfaces should be explicit and expose all information related to their events.

We believe that the techniques we have described in this paper can provide a substantial benefit to users of existing operating systems, enabling a viable third-party industry for developing and deploying operating system extensions. The resulting competition will stimulate innovation and increase the rate of technology transfer from operating systems research into production systems.

Availability

Current status and source code are available at `http://now.cs.berkeley.edu/Slic/`.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 USENIX Summer Conference*, pages 93–112, June 1986.

[2] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User-Level: the Ufo Global File System. In USENIX, editor, *Proceedings of the 1997 USENIX Conference*, pages 77–90, January 1997.

[3] C. Amza, Alan L. Cox, Sandhya Dwarkadas, Peter Keleher, H. Lu, R. Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] Arindam Banerji, John M. Tracey, and David L. Cohn. Protected Shared Libraries — a new approach to modularity and sharing. In *Proceedings of the 1997* USENIX *Technical Conference*, pages 59–76, January 1997.

[5] B. N. Bershad, S. Savage, E. G. Sirer P. Pardyak, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Calls. In *ACM Transactions on Computer Systems*, pages 37–54, February 1990.

[7] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, October 1994.

[8] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, October 1997.

[9] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[10] Chaos Computer Club. CCC: Microsoft security alert. http://berlin.ccc.de/radioactivex.html, March 1997.

[11] Apple Computers. *Inside Macintosh, Macintosh Toolbox Essentials*. Addison-Wesley, 1992.

[12] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice and Experience*, 21(8):757–85, August 1991.

[13] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, 1996.

[14] D. R. Engler, M. F. Kaashoek, and Jr J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[15] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[16] Simon Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly and Associates, Sebastopol, CA, first edition, December 1994.

[17] Ian Goldberg, David Wagner, Randy Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[18] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N. Bershad. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[19] IBM Virtual Machine Facility /370 Planning Guide. Technical Report GC20-1801-0, IBM Corporation, 1974.

[20] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[21] Yousef Khalidi and Michael Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, December 1993.

[22] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[23] Steven R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 Summer* USENIX *Technical Conference*, pages 238–247, 1986.

[24] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized overlaying. In *Proceedings of the 1992 USENIX Winter Conference*, January 1992.

[25] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[26] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 229–243, October 1996.

[27] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[28] John Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the 1990 Summer* USENIX *Conference*, pages 247–256, June 1990.

[29] Przemysław Pardyak and Brian N. Bershad. Dynamic binding in an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, October 1996.

[30] Dennis M Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[31] R. Sandberg, D. Goldberg, Steven Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985* USENIX *Technical Conference*, pages 119–130. Sun Microsystems, June 1985.

[32] Andrew Schulman, Raymond J. Michels, Jim Kyle, Time Paterson, David Maxey, and Ralf Brown. *Undocumented DOS.* Addison-Wesley, 1990.

[33] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[34] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Issues in extensible operating systems. Technical Report TR-18-97, Harvard University, 1997.

[35] Emin Gün Sirer, Stefan Savage, Przemysław Pardyak, Greg DeFouw, Mary Ann Alapat, and Brian N. Bershad. Writing an operating system using Modula-3. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[36] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard, October 1994.

[37] David C. Steere, James J. Kistler, and M. Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Proceedings of the 1990 USENIX Summer Conference*, pages 325–331, June 1990.

[38] Computer Emergency Response Team. Ghostscript Vulnerability. CERT Advisory CA-95.10, CERT, August 1995.

[39] Computer Emergency Response Team. Vulnerability in expreserve. CERT Advisory CA-96.19, CERT, August 1996.

[40] Computer Emergency Response Team. Vulnerability in Solaris admintool. CERT Advisory CA-96.16, CERT, August 1996.

[41] Computer Emergency Response Team. Vulnerability in WorkMan. CERT Advisory CA-96.23, CERT, October 1996.

[42] Computer Emergency Response Team. Vulnerability in talkd. CERT Advisory CA-97.04, CERT, January 1997.

[43] Marvin Theimer, K. Landtz, and David Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[44] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.

[45] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[46] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[47] Nick Wingfield. ActiveX used as hacking tool. `http://www.news.com/News/Item/0%2C4%-2C7761%2C00.html`, February 1997.

[48] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–344, June 1974.

[49] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[50] G. Pascal Zachary. *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft.* Macmillan, Inc., 1994.

[51] Sognian Zhou, Jingwen Wang, Xiaohn Zheng, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.

A Transactional Memory Service in an Extensible Operating System

Yasushi Saito and Brian Bershad
{yasushi,bershad}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper describes *Rhino*, a transactional memory service implemented on top of the *SPIN* operating system. *Rhino* is implemented as an extension that runs in *SPIN* kernel's address space. We discuss how the extension structure of *Rhino* can solve performance problems previously unavoidable in traditional systems, and we quantify its benefits. We also introduce three alternative buffer management schemes and study their performance under various workloads.

1 Introduction

This paper describes *Rhino*, a transactional memory service that lets applications perform transactions on a virtual memory through ordinary loads and stores.

Rhino runs on the *SPIN* operating system, which lets user applications modify and augment the kernel by safely downloading code (*extensions*) into the kernel address space [2]. *Rhino* is implemented as an extension. Extensions can efficiently manipulate kernel resources such as virtual memory pages and files, and can thus avoid the performance bottlenecks that plague traditional systems.

We have developed several versions of *Rhino* as our understanding of *SPIN* and its target applications grew. These versions share the same transaction mechanisms, but they differ in the way they detect updates to memory and manage buffers. In this paper, we explore the tradeoffs among these systems under different workloads. We also show how *SPIN*'s extension architecture enhances the performance of *Rhino* by comparing *Rhino* with its user-space, UNIX-based implementation.

1.1 Transactional Memory Requirements

Transactional memory service allows applications to access a database file mapped onto virtual memory regions in an atomic, isolated, and durable (also known as *ACID*) manner [8]. It is used as a runtime engine for managing persistent, pointer-rich data structures, such as graphic design databases, project management databases, and directory services.

Transactional memory is similar to a memory-mapped file, but the ACID property requires additional operating system support. To ensure atomicity, wherein a set of database accesses must be performed in an all-or-nothing manner, transactional memory records all updates to the database; it then replays or unwinds them after the system crashes or when the application aborts an operation. To isolate applications, transactional memory must lock virtual memory regions that the applications have accessed.

1.2 Limitations of Existing Systems

The functions provided by conventional operating systems do not meet the requirements of transactional memory systems. The following sections highlight three areas where existing operating systems respond poorly to the needs of transactional memory.

1.2.1 Write Detection

Transactional memory service must keep track of the database changes to ensure ACID property. In user-space transactional memory implementations, writes to the database are usually detected by the MMU protection and upcalls from the operating system (`SIGSEGV` signals in UNIX). However, such

implementations incur high overhead [21], as shown in Figure 1 (a). When a page fault occurs, the kernel performs a full context switch into the signal handler. The signal handler calls the server to bring the faulted page into client memory. It then issues a system call, such as `mprotect`, to change the MMU protection and makes a context switch back to the faulted context. The whole process requires at least eight user-kernel boundary crossings and four context switches.

1.2.2 Inter-process Communication

Most transactional memories are organized as client-server systems, since they can easily handle concurrent updates by multiple applications. These systems require frequent inter-process communication (IPC) for data fetching, database locking, and transaction control. IPC performance becomes more critical in a transactional memory services where clients and the server exchange data pages, than in relational database systems where only queries and answers are exchanged. Although fast IPC mechanisms have been studied extensively [1, 11], they have not made their way into mainstream operating systems. Thus, IPC in existing operating systems is slow, and it often becomes the bottleneck in transactional memory implementations.

1.2.3 Buffer Management

A transactional memory service must often handle databases whose size exceeds that of main memory. When the demand for memory exceeds a limit, the operating system evicts pages from applications. When a database buffer page held on ordinary virtual memory is evicted, extra disk accesses are required to swap out and later swap in the page. Straightforward implementations have been unable to address this problem, called *double paging* [13].

1.2.4 Organization of the Paper

Section 2 introduces the *SPIN* operating system. Section 3 reviews SPIN's *Rhino* extension. In Section 4, we discuss implementation issues and describe the advantages and disadvantages of three alternative buffer management schemes. *Rhino*'s performance is contrasted with UNIX implementations in Section 5. Section 6 examines works related to ours. We summarize our findings in Section 7.

2 Overview of *SPIN*

SPIN is an extensible operating system. Applications can safely extend the kernel functionality by downloading code into the kernel address space [2]. The *SPIN* operating system consists of the *kernel*, which provides basic services such as CPU scheduling and device management, and *extensions*, which are downloaded into the kernel address space after the kernel boots. Extensions efficiently communicate and share resources with the kernel and other extensions; from a performance viewpoint, they resemble dynamically linked modules.

The *SPIN* kernel and extensions are written in Modula-3 [14], a general purpose, typesafe language. The Modula-3 compiler, the *SPIN* runtime environment and the dynamic linker ensure that extensions cannot arbitrarily access memory or other critical resources, such as I/O ports and interrupt masks.

2.1 Use of Extensions in *SPIN*

Figure 2 shows a typical configuration of *SPIN*. Some *SPIN* extensions provide basic services used by other extensions. For example, file system extensions provide common directory and file operations like those found in UNIX. The virtual memory extension provides address spaces and memory objects similar to those found in Mach [23].

SPIN also supports user-space applications. Although the *SPIN* kernel does not support native system calls, it provides mechanisms that allow extensions to catch events from user-space applications, such as system calls and page faults. The UNIX emulation extension uses these mechanisms to provide the UNIX API for user-space applications[16]. *Rhino* is another extension that implements a transactional memory service for user-space applications. Note that unlike extensions, user-space applications can be written in any language. They are protected from other components by hardware mechanisms, as they are in other operating systems.

2.2 *SPIN*'s Approach to Transactional Memory

Rhino is implemented as an in-kernel extension. This design makes it possible to overcome the operating system deficiencies described in Section 1.1. For example, Figure 1 (b) shows page fault handling in *Rhino*. By placing the page fault handler inside the kernel space, *Rhino* can process a write detection event with two user-kernel crossings and zero

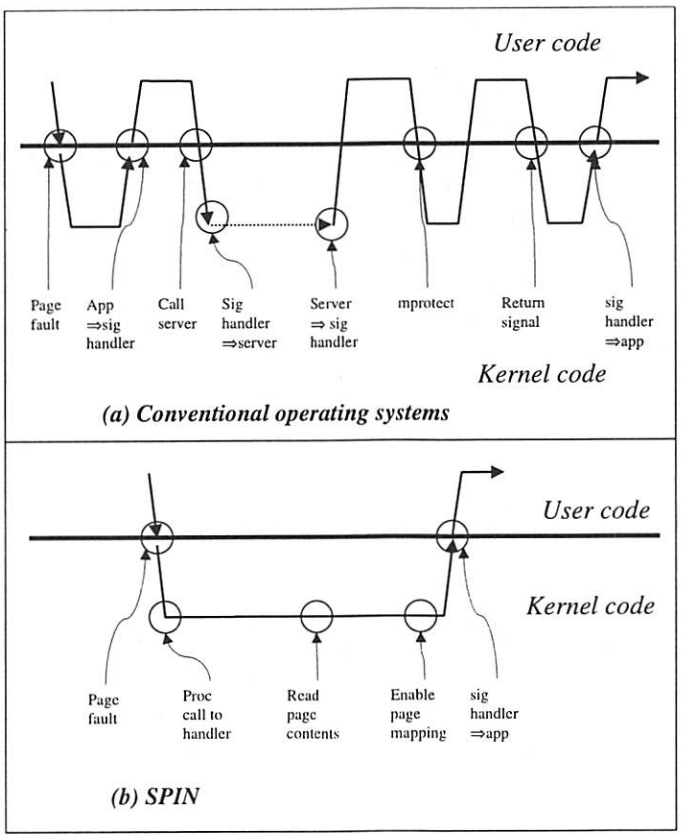


Figure 1: Page fault handling comparison. Conventional operating systems require at least eight user-kernel boundary crossings and four context switches to read the page contents in response to a page fault. In contrast, *SPIN* requires two user-kernel boundary crossings and no context switches.

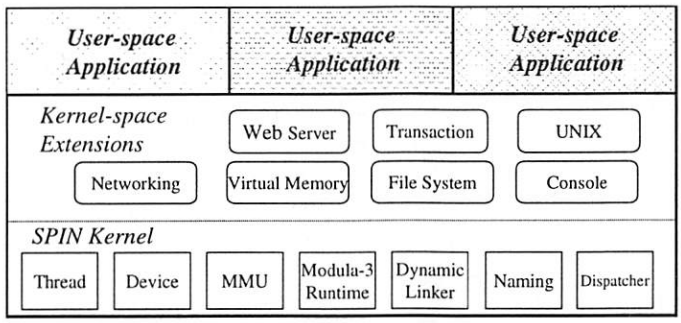


Figure 2: A typical *SPIN* configuration. The *SPIN* kernel and extensions are in the kernel address space. Each user-space application runs in its own address space (shown by different stipple patterns).

context switches.

Extensions also reduce the IPC overhead that exists in client-server systems. Typical IPC involves argument copying, context switching to the server (assuming the server runs on the client machine), writing back of results, and context switching back to the client. In *Rhino*, context switches are eliminated, since the extension is in the kernel address space and shared by all user-space applications.

Another example of the benefits of *SPIN*'s extension approach is buffer management. *Rhino* maps the database buffer directly onto an application's address space. It cooperates with the virtual memory extension to swap buffer pages directly to a database file rather than to a disk, thus solving the double paging problem.

3 Overview of *Rhino*

Rhino is structured as an extension that communicate with user-space applications via system calls. This section reviews the structure of the *Rhino* extension and its usage.

3.1 *Rhino* Structure

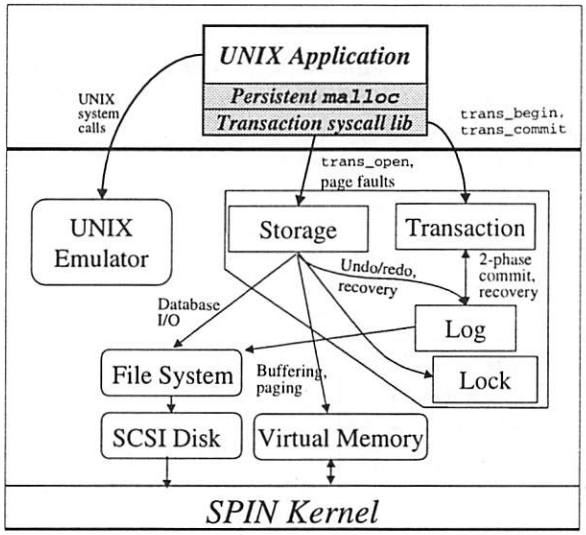


Figure 3: The *Rhino* extension structure. The *Rhino* system consists of shaded regions. It uses the file system extensions for database I/O and logging, and the virtual memory extensions for database buffering. *Rhino* is used by UNIX applications linked with the transaction library.

The shaded regions of Figure 3 show the structure of *Rhino*. Persistent `malloc` and system call stubs are linked into applications as a library. The following additional components reside in the kernel address space as an extension.

- The *transaction manager* starts and terminates transactions.
- The *storage manager* is the core of *Rhino*.[1] It detects applications' reads and writes and logs them to ensure the ACID property. It also manages buffers using memory objects.
- The *log manager* manages the log device, which is a sequential-write, random-read persistent device [8]. It is used by the storage manager to guarantee database durability and atomicity. The log manager also coordinates crash recovery when *Rhino* is installed.
- The *lock manager* manages locks on regions. It also detects deadlocks.

Rhino uses several standard extensions to carry out operations. It stores databases and log records in files managed by file system extensions. Files are usually stored in the extent-based file system, which allocates files on contiguous blocks and does not cache blocks in memory. *Rhino* cooperates with the virtual memory extension to manage database buffers. Its buffer management is described in detail in Section 4.

3.2 Application Programming in *Rhino*

Database files managed by *Rhino* are accessed either by user-space applications or by other in-kernel extensions. For ease of use and understanding, we use C nomenclature to present interfaces accessed by user-space applications.

Table 1 shows the system calls supported by *Rhino*. The system call interface resembles those found in other systems [17, 10, 20]. Figure 4 shows a simple application that writes "z"s onto the region that extends from byte 256 to byte 384 in the file "`/efs/test`". To access a database file, the application first calls `trans_open` to get a handle to the file. It then calls `trans_mmap` to map the file onto a virtual address space region (that must be MMU-page aligned). After setup of the transactional memory region, the application need

[1] Note that our use of the term "storage manager" differs from that in other database literature [8]. "Storage manager" commonly refers to the code that manages raw disk I/O. In our system, the storage manager is a resource manager specialized for file manipulation. Raw disk I/O is not part of the transaction service, since *Rhino* directly uses the raw I/O facility provided by *SPIN*'s file system.

only demarcate transactions by using `trans_begin`, `trans_commit`, and `trans_abort`. Accesses to the transactional memory region are detected through page faults.[2]

```
main()
{
    tid_t tid;
    sid_t sid;
    void *base = (void*)0x10000;

    /* Open the file /efs/test. */
    sid = trans_open("/efs/test");
    /* Map the storage from 0x10000 .. 0x90000. */
    trans_mmap(base, 0x80000, sid);

    tid = trans_begin();
    /* Fill the region with "z". */
    memset(base + 256, 'z', 128);
    if (something_wrong()) trans_abort(tid);
    else trans_commit(tid);
}
```

Figure 4: Sample application in user space.

4 Implementation Issues

This section describes the *Rhino* functions needed to implement a transaction. Buffering is the key to high performance. Locking and write detection are needed to ensure the ACID property [8].

4.1 Buffering

Rhino stores database contents in memory-mapped files; in other words, the database is buffered on pages that are mapped directly onto an application's address space. Memory-mapped files thus avoid the double paging problem even if there is memory competition.

Efficient buffering must comply with the write ahead logging (*WAL*) rule, which dictates that whenever a page is to be evicted, its log records (undo records) must be flushed to the log device [8, 13]. *Rhino* ensures WAL by implementing its own pageout procedure. Whenever the kernel chooses a page for purging, the *Rhino* pageout module is called to flush the log into the log device.

4.2 Locking

Rhino lets multiple applications access a database concurrently. To ensure isolation among transactions, database regions must be locked until a transaction finishes. *Rhino* asserts locks in MMU page grain using page faults.

[2] One version of *Rhino* requires a `trans_setrange` system call instead of page fault detection. In that version, "`trans_setrange(sid, 256, 128);`" is needed just before memset.

Before a transaction starts, MMU mappings for the database region are invalid. The first access to a page causes a page fault. The page fault handler in the *Rhino* extension obtains either a read or a write lock on the page, depending on whether the access is load or store. This scheme is essentially the same as that used in systems such as ObjectStore [10] and QuickStore [22]. Other locking approaches are possible, such as requiring applications to issue a system call to lock a region [7]. However, we decided on the MMU-based automatic locking approach to make the programming interface as simple as possible.

A shortcoming of this approach is that multiple threads in a single process cannot execute transactions simultaneously on the same database. However, this is not a serious problem since thread is not a unit of protection in most operating systems including *SPIN*; protecting database accesses by threads does not provide much help to programmers.

4.3 Write Detection

Transactional memory service must detect and log all writes to the persistent region, permitting changes to the database to be undone or redone atomically [8]. We implemented three versions of write detection in *Rhino* to study their performance trade-offs under various workloads. They are *setrange*, *page grain logging*, and *page diffing*. Setrange requires applications to issue a `trans_setrange` system call before modifying the database. The other two versions rely on the MMU to detect writes and differ in detection precision. Page grain logging treats the whole page as modified when at least one byte on the page changes. Page diffing tries to compute the exact set of modifications by comparing old and new page contents.

4.3.1 Setrange

The setrange approach creates a memory object for each open database file. It is mapped to the application's address space when `trans_mmap` is called. Before modifying a region in the database, the application must issue the `trans_setrange` system call to notify the *Rhino* extension about the region. In response to the call, *Rhino* pins down all buffer pages in the region so they will not be paged out until the transaction ends. It then records the region in a per-transaction record. Upon commit, *Rhino* scans the per-transaction record and logs the contents of each region as redo records. Thus, it implements a

System Call	Function
`storage = trans_open(path)`	Opens the database file *path*
`trans_close(storage)`	Closes the file
`trans_setrange(storage, from, len)`	Notifies the modification to *Rhino*. This system call exists only in one of the three alternative versions of buffer management.
`trans_mmap(addr, length, storage)`	Maps the file onto caller's address space
`trans_munmap(addr, length)`	Unmaps the file
`trans_id = trans_begin()`	Begins a transaction
`trans_commit(trans_id)`	Commits the transaction
`trans_abort(trans_id)`	Aborts the transaction and rolls back its effect

Table 1: *Rhino* API

no-steal, no-force policy [6]. Update detection using `setrange` was first implemented in RVM [17].

4.3.2 Page Grain Logging

Instead of relying on system calls from applications, page grain logging version uses MMU protection to detect writes. Database contents are stored in a memory object, as in the setrange version. All virtual memory mappings for the memory object are invalid before a transaction starts.

When the application writes onto the memory object, a page fault occurs, and the *Rhino* storage manager brings page contents in from disk, if necessary. The current contents of the page are then logged immediately as an undo record. Finally, the storage manager maps the page onto the application's address space. Upon commit, contents of all modified pages are logged as redo records.

When a buffer page is chosen as a pageout victim, the storage manager flushes the undo records generated for the page. Next, it writes the contents of the storage page into the database file. Finally, it removes the page from the memory object. Thus, this version implements a steal, no-force buffer management. Variations of page grain logging can be found in many transactional memory systems, including ObjectStore [10].

4.3.3 Page Diffing

Page diffing resembles page grain logging. However, it tries to reduce the size of the log by computing differences between old and new page contents. When a database file is opened, the storage manager creates two memory objects. The *storage object* is mapped onto the application's address space, and it caches the up-to-date contents of the file. The *shadow object* holds old buffer contents. It is not mapped onto address spaces; rather, it is used only to group pages together. Figure 5 shows how the two memory objects are used.

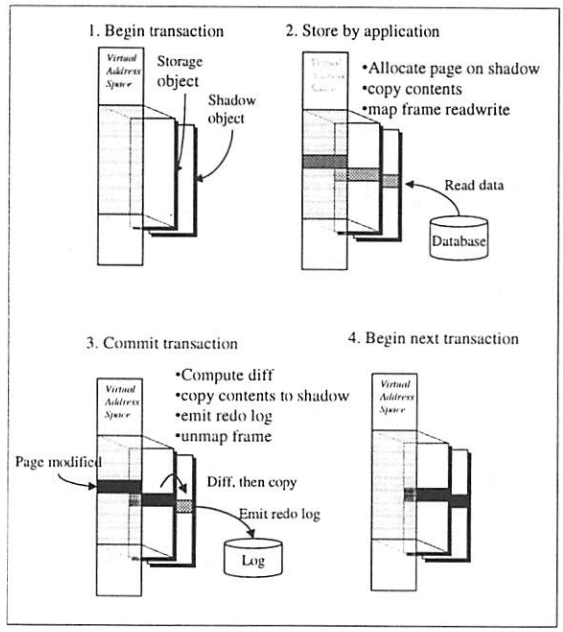


Figure 5: Page diffing algorithm

Before each transaction, all MMU mappings for the database region are invalid. In response to a page fault caused by an application store, the storage manager brings the page into the storage object, if necessary. It then allocates a page on the shadow object and copies the contents of the storage object page onto the new page. Finally, the storage object page is mapped onto the application's address space.

Upon commit, it compares the contents of each modified page and its shadow word by word and computes the differences between them. The "diff"s are logged as redo records. After commit, all mappings for the database region are invalidated.

When either a storage page or its shadow is chosen as a pageout victim, and the storage page is modified by some transaction, *Rhino* computes a "reverse diff" of the shadow and the storage page. The reverse diff is logged and flushed as the undo record. Next, the contents of the storage page are written out to the database file. Finally, both the storage page and the shadow page are removed from the memory objects. When the storage page is dirty but is not being modified by a transaction, *Rhino* writes the page to the database file without generating an undo record.

With this design, if the number of pages accessed by the transaction is smaller than the main memory size, no undo records are generated. Undo records are generated only when buffer pages are evicted.

Page diffing was first proposed in QuickStore [22]. Unlike *Rhino*, QuickStore does not allow dirty pages to be flushed before commit (no-steal policy). Thus, QuickStore does not generate reverse diffs.

4.3.4 Trade-offs among Write Detection Versions

We implemented the setrange version first because it was easy to understand and fast for small transactions since it can minimize the amount of redo records. However, we found a number of disadvantages when we ran bigger transactions.

- Setrange requires manual intervention.

  Programmers have to call `trans_setrange` manually before modifying a database region. Forgetting to make this call could result in unrecorded the changes to the database.

- The overhead of calling `trans_setrange` is significant in big transactions.

  When a transaction makes many `trans_setrange` calls, call overhead becomes sizeable. Note that this problem is less serious in systems like RVM [17] and Vista [12], which implement setrange as a library.

- Memory overhead increases in large transactions.

  The biggest problem with setrange is that each setrange region must be recorded until the transaction terminates. Thus, we cannot limit the amount of memory required to keep track of a transaction.

  A related problem is that setrange does not work well with the steal buffer management policy, which is essential to run large transactions. Because ranges must be remembered in memory until a transaction commits, it is difficult to evict pages in the middle of a transaction.

Page grain logging can limit memory consumption regardless of the amount of modifications: all modifications are recorded in memory object pages, and pages can be purged. It is less error prone than setrange, because it does not require programmer cooperation. Another advantage is that it can amortize the write detection cost when many bytes are updated in a page, because detection is required only once[3]. Thus, it is faster than setrange when many bytes are modified per page.

The problem with this approach is the log can grow quite large. For each modified page, log records twice as large as the page size are generated (one for an undo record, one for a redo record). This is wasteful for two reasons. First, it generates undo and redo records for the whole page even if only a single byte is modified on the page. Second, page grain logging blindly generates undo records even for transactions small enough not to require paging, in which case undo records are not needed [8].

Page diffing combines the advantages of setrange and page grain logging. It shares all the advantages of page grain logging. In addition, it can minimize log size by computing page diffs.

However, page diffing introduces overhead that did not exist in earlier versions. One source of overhead is the page diffing itself. Page diffing needs to walk over two pages and write out differences to another memory region. Not only is this procedure slow, but it retards other procedures by contaminating the CPU cache. Another source of overhead is the memory pressure imposed by shadow pages. In the worst case, one in which all accessed pages are modified, the effective memory size is halved. Thus, the system will have more paging activities.

5 Performance

This section evaluates the performance of *Rhino*. All measurements were carried out on a DEC Alphastation 250 with a 21064A CPU running at 266MHz, 47MB of user memory, and a DEC RZ26L 1GB SCSI disk.

We compared five systems: the three versions of

[3] When the page is purged and later brought in again, *Rhino* takes a page fault again.

Rhino running on *SPIN*, the page diffing version of *Rhino* running on Digital UNIX 3.2, and ObjectStore 4.0 running on Digital UNIX 3.2 [10].

Rhino on Digital UNIX uses the same page diffing code to detect modifications, but buffers are on ordinary virtual memory pages instead of a memory-mapped file, and page faults are detected using UNIX signals. Digital UNIX *Rhino* was measured to quantify the benefits of *SPIN*'s extension architecture, which allows low-cost communication between extensions and the kernel.

ObjectStore is a client-server database management system that buffers database contents on a client's virtual memory. It implements no-steal, no-force buffer management and page grain logging. ObjectStore is included to compare *Rhino* against a state-of-the-art, object-oriented database system.

We first present the micro-benchmarks that show the latency of the critical paths. Next, we present results from two benchmarks, RVM [17] and OO7 [3]. The RVM benchmark typifies small update transactions, while OO7 typifies graphical CAD database operations.

5.1 Micro-benchmarks

This section compares the micro-benchmark performance of *SPIN*-based *Rhino* and Digital UNIX-based *Rhino* to show how the extension architecture of *SPIN* improves the performance of critical functions. Table 2 shows the time breakdown of some important events.

Null call indicates a null system call overhead (on Digital UNIX, we measured the latency of `getpid`). *SPIN* is slower than Digital UNIX, because the implementation of system call in *SPIN* requires the use of additional mechanisms to protect the kernel from the runtime failure of an extension [16].

Begin shows the latency of `trans_begin`. *Commit(ro)* is the time to commit a read-only transaction. *Commit(8byte)* is the time to commit a transaction that modified 8 bytes on a single page. Page diffing is used during commits.

Four numbers are shown for page faults. "Read" faults are caused by load instructions, and "write" faults by store instructions. "Warm" faults occur when database contents are in main memory. Thus, these are times with no disk I/O. "Cold" faults occur when database contents are not in main memory and require pages to be read from the disk.

The *SPIN* version outperforms the UNIX version for all events except the null call. The performance difference is largest for warm page faults. There are two reasons for this: (1) since the page fault handler in *SPIN* runs in the kernel address space, it can eliminate most of the user-kernel crossings, and (2) page table manipulation in *SPIN* is more efficient than `mprotect` used in Digital UNIX. In *SPIN*, MMU can be manipulated by rewriting the MMU page table directly. On the other hand, `mprotect` requires more work, because it must manipulate the memory object map data structure to make its effect persist regardless of paging activity.

5.2 RVM Benchmark

The RVM benchmark is a program developed by the authors of RVM [17]. Each transaction reads and updates three 128-byte blocks and appends one 64-byte block to the end of the database. One 128-byte block is chosen randomly from the entire database; the other two 128-byte blocks are chosen from a narrow region. Thus, this benchmark measures the performance of small transactions.

We varied the database's size, ran 4000 transactions for each size, and calculated the mean time needed to complete one transaction. The number of bytes modified by each transaction does not depend on the database size. However, transactions running on small databases can utilize buffers more efficiently when many transactions are run successively.

Figure 6 shows the results. For small databases, setrange and page diffing perform almost equally well. However, as the database's size grows, the performance of page diffing drops quickly: the page diffing algorithm can utilize only half the amount of main memory available to the other schemes, since all the bytes accessed are modified in this benchmark. Page grain logging does a little worse than setrange for all database sizes because of increased logging activities. The UNIX page diffing version consistently performs about 1.5 to 2 times more slowly than *SPIN*'s page diffing. This difference is due to increased user-kernel crossings and extra data copying during I/O, because buffer pages are in ordinary virtual memory. ObjectStore fares badly in this benchmark. Since it performs page grain logging, whole page contents must be communicated to the server via IPC. Thus, it is not suited to small transactions.

5.3 The OO7 Benchmark

OO7 is the standard benchmark for object-oriented databases [3]. The database consists of objects of

Event	UNIX			SPIN		
	total	trap	other	total	trap	other
null call	2.14	2.14	–	6.11	6.11	–
begin	55.4	–	–	26.4	9.4	14
commit (ro)	152.3	–	–	29.7	13.4	16.3
commit (8byte)	14200	–	–	13328	15.2	13313
page fault (read, warm)	282.4	134	148.4	55.3	13.5	41.8
page fault (write, warm)	234.3	133	101.3	68.8	16.5	52.3
page fault (read, cold)	2881	131	2750	2272	20	2252
page fault (write, cold)	3059	113	2946	3054	19.7	3034

Table 2: Comparison of critical path latencies. The *total* columns show the total microseconds spent in each event. *Trap* columns show the overhead needed to pass control to the signal handler (on Digital UNIX) or to the *Rhino* page fault handler (on *SPIN*). Begin and commit for the UNIX implementation are implemented in the user space, and they issue multiple system calls. Thus, only the total elapsed times are shown for them.

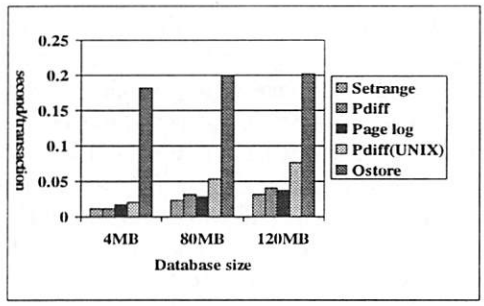


Figure 6: RVM benchmark results. 4000 transactions were run consecutively. The time to complete a single transaction is shown for each system.

various types connected in a tree structure (Figure 7).

We used the small, medium-3, and medium-9 configurations described in [3] (numbers in the medium configurations show the density of internal connections). The total database sizes are about 10MB for the small, 60MB for the medium-3, and 100MB for the medium-9.

We ran three types of traversals, *T1*, *T2A*, and *T2C*. They all traverse the object hierarchy and visit one element within each intermediate node. T1 visits all the elements in read only mode. T2a updates one element for each intermediate node. T2c updates each element four times.

Two types of numbers, *cold* and *warm*, are shown for the small configuration[4]. To obtain the cold numbers, we started the benchmark with an empty buffer cache[5]. An ObjectStore file open call (`objectstore::open`) pre-fetches some of the database contents into memory. The time needed to execute this procedure is also included in the cold numbers. Hot numbers are obtained by running four consecutive transactions after the cold run and computing the mean of the first three. For ObjectStore, the option to retain persistent pointers (`objectstore::retain_persistent_pointers`) was enabled. Thus, hot runs do not include pointer-swizzling overhead.

We do not report the setrange performance, because the setrange algorithm could not run OO7 for larger databases: as described in Section 4.3.4, setrange must retain all the range information in memory until a commit, and it uses up the in-kernel heap.

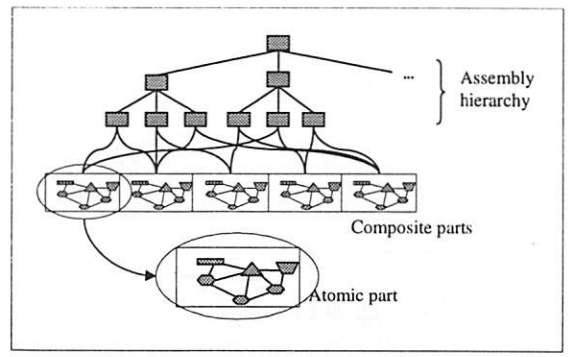


Figure 7: OO7 database structure. The database consists of composite parts, each of which is a web of atomic parts. Composite parts are indexed by a tree.

[4] Warm numbers are not shown for the medium configurations. In fact, they are almost same as their cold counterparts.

[5] We used a raw device for databases to bypass operating system caching.

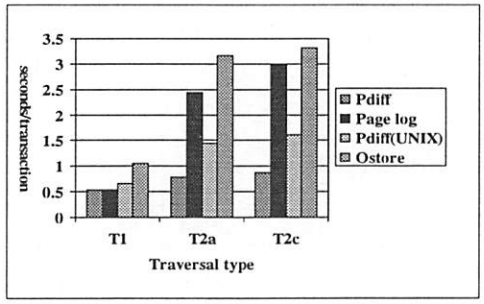


Figure 8: Small configuration results. Buffer cache was warm.

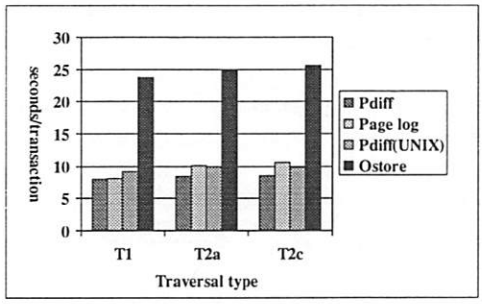


Figure 9: Small configuration results. Buffer cache was cold.

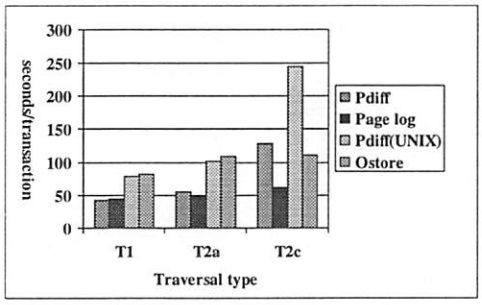


Figure 10: Medium configuration (fanout=3) results. Buffer cache was cold.

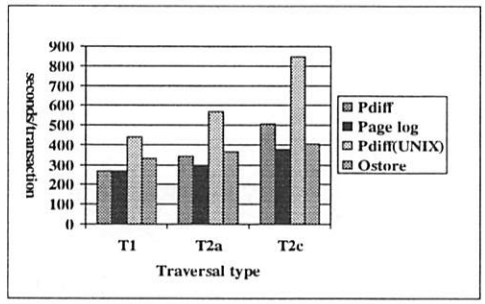


Figure 11: Medium configuration (fanout=9) results. Buffer cache was cold.

Figure 8 shows "warm" results from the small database. First, for T1 traversal, page grain logging and page diffing exhibit the same performance, because they perform the same tasks in read-only transactions. For other traversals, page diffing performs better than page grain logging because of smaller log activity. The UNIX page diffing version is consistently twice as slow as *SPIN*'s page diffing version, for the same reason described in the previous section. ObjectStore consistently performs worse than any of the *Rhino* versions. However, the performance discrepancy is smaller than in the RVM benchmark, because OO7 touches more bytes per page, and thus IPC overhead is amortized.

Figure 9 shows "cold" results for the small database. In the cold runs all *Rhino* versions perform about the same, because disk I/O dominates the time.

Figures 10 and 11 show the "cold" results on a medium database of fanout 3 and fanout 9, respectively. Page diffing does not perform well in these benchmarks, because of the memory pressure caused by shadow pages. Page grain logging is clearly the best choice in medium-3 and medium-9. ObjectStore performs better here, because the IPC cost is amortized over a large amount of updates.

5.4 Summary

This section compared the three versions of *Rhino* with a transactional memory service implemented entirely in user-space. The micro-benchmark numbers show that the extension structure of *Rhino* improves the latency of all the events critical to performance, especially the page fault handling. The two application benchmarks, RVM and OO7, compare trade-offs among the three *Rhino* buffer management alternatives. Page diffing version runs efficiently in small transactions. It can run larger transactions, but its performance suffers from memory

under-utilization due to shadow pages. Page-grain logging does worse for small transactions due to large log size, but it scales well up to large transactions. The setrange version is fastest in small transactions, but it does not scale for large transactions. The RVM and OO7 results also show that *SPIN*-based *Rhino* outperforms user-space transactional memory implementations.

6 Related Work

This section reviews systems related to *Rhino*. We include in our review object-oriented databases and persistent languages, since they are closely related to transactional memory.

ObjectStore [10], QuickStore [22], and QuickSilver [18] are client-server systems in which a server process manages transactions, and clients perform IPC to the server to access database contents. The advantage of this approach is that servers can transparently support clients running on different hosts. However, this approach also means that even local clients must communicate through a slow IPC channel, thus creating performance problems.

RVM [17] and Texas [20] are implemented as a library that is linked into user-space applications. Since they have no IPC overhead, unlike client-server systems, they can be fast. However, they are inherently single-user database systems, because there is no single authority that allows safe data sharing.

A problem common to the systems discussed thus far is double paging, since they are implemented as ordinary user-space applications. The only way to solve the double paging problem is to reserve a fixed amount of memory for the database management system and let the system perform its own paging. This approach is effective when the whole machine is dedicated to the database service. However, when there are other applications competing for memory, which is typical in transactional memories, this solution does not work well.

Some systems try to solve the double paging problem by using memory-mapped files and a special system call that lets user programs control the way pages are evicted. RPVM [5] adds a system call that dictates the order of page eviction. By telling the kernel to purge a buffer page after it purges log pages that record updates to that page, write ahead logging (WAL) can be implemented. Camelot [7] and Cricket [19] use the Mach external pager mechanism [23] to implement WAL. One difference between these systems and *Rhino* is that the former are user-space applications. Thus, they cannot avoid overhead due to a large number of user-kernel crossings.

Vista[12] uses Rio, a non-volatile file buffer, to implement transactions. Rio makes all updates to the buffer permanent immediately by recovering buffer contents during the system reboot. Thus, Vista transactions are orders of magnitude faster than transactions based on disk-logging.

Finally, there are systems that implement transactions inside the kernel. IBM CPR [4] and Pilot [15] support transactional updates of memory-mapped files. These systems solve problems found in other systems. However, most applications do not use transactions frequently enough to afford the complexity introduced by embedding transaction support in the kernel, making this approach uneconomical.

Herlihy and Moss proposed a transactional memory that is a CPU instruction set designed to support atomic memory updates[9]. Our use of the term is not related to theirs.

7 Conclusions

This paper described the implementation and performance of *Rhino*, a transactional memory implemented on the *SPIN* operating system. By implementing *Rhino* as an extension dynamically loaded into the kernel address space, we avoid problems associated with traditional systems, such as double paging and user-kernel boundary crossing overhead.

We implemented three write-detection approaches (setrange, page grain logging, and page shadowing) to study their trade-offs in the extension environment. Performance measurement demonstrate that all versions of our system outperform user-space implementations. Also, among the three variations of *Rhino*, it was found that setrange and page diffing perform equally well for small transactions. Page grain logging performs well for large transactions.

The *SPIN* operating system, as well as the *Rhino* transactional memory service described in this paper, can be obtained via the world wide web at http://www.cs.washington.edu/research/projects/spin.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM TOCS*, 8(1):37–55, February 1990.

[2] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM SOSP-15*, Copper Mountain, CO, December 1995.

[3] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *SIGMOD Conference Proceedings*, pages 12–21, Washington D.C., June 1993. ACM.

[4] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM TOCS*, 6(1):28–50, January 1988.

[5] Khien-Mien Chew and Avi Silberschatz. Toward Operating System Support for Recoverable-persistent Main Memory Database Systems. Technical Report CS-TR-92-05, University of Texas at Austin, September 1992.

[6] Wolfgang Effelsberg and Theo Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[7] Jeffrey Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon.* Morgan Kaufmann, San Francisco, CA, 1991.

[8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Francisco, CA, 1993.

[9] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ICSA*. ACM, 1993.

[10] Object Design Inc. home page. http://www.odi.com.

[11] Jochen Liedtke. On Micro-Kernel Construction. In *ACM SOSP-15*, Copper Mountain, CO, December 1995.

[12] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *ACM SOSP-16*, 1997. See also http://www.eecs.umich.edu/~pmchen/.

[13] Dylan McNamee. *Virtual Memory Alternatives for Transaction Buffer Management in a Single-Level Store.* PhD thesis, University of Washington, November 1996.

[14] Greg Nelson, editor. *Systems Programming in Modula-3.* Prentice Hall, 1991.

[15] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *CACM*, 23(2):81–92, February 1980.

[16] Yasushi Saito and Brian Bershad. System Call Support in an Extensible System. See http://www.cs.washington.edu/homes/yasushi, September 1997.

[17] M. Satyanarayanan, Henry Mashburn, Puneet Kumar, David Steere, and James Kistler. Lightweight Recoverable Virtual Memory. *ACM TOCS*, 12(1):33–57, February 1994.

[18] Frank Schmuck and Jim Wyllie. Experience with Transactions in QuickSilver. In *SOSP-13*, pages 239–253, October 1991.

[19] Eugene Shekita and Michael Zwillling. Cricket : A Mapped, Persistent Object Store. Technical Report TR-956, University of Wisconsin, 1990.

[20] Vivek Singhal, Sheetal Kakkad, and Paul Wilson. Texas: An Efficient, Portable Persistent Store. In *Proc. Fifth International Workshop on Persistent Object Systems*, pages 11–33, September 1992.

[21] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *ASPLOS 6*. ACM, October 1994.

[22] Seth J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems.* PhD thesis, University of Wisconsin, September 1994.

[23] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *ACM SOSP-11*, pages 63–76, Austin, TX, November 1987.

Dynamic C++ Classes

A lightweight mechanism to update code in a running program

Gísli Hjálmtýsson
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932
gisli@research.att.com

Robert Gray
Thayer School of Engineering[1]
Dartmouth College
Hanover, NH 03755
rgray@cs.dartmouth.edu

Abstract

Techniques for dynamically adding new code to a running program already exist in various operating systems, programming languages and runtime environments. Most of these systems have not found their way into common use, however, since they require programmer retraining and invalidate previous software investments. In addition, many of the systems are too high-level for performance-critical applications. This paper presents an implementation of *dynamic classes* for the C++ language. Dynamic classes allow run-time updates of an executing C++ program at the class level. Our implementation is a lightweight proxy class that exploits only common C++ features and can be compiled with most modern compilers. The proxy supports version updates of existing classes as well as the introduction of new classes. Our language choice and proxy implementation is targeted towards performance-critical applications such as low-level networking in which the use of C++ is already widespread.

1. Introduction

Every modern organization has software systems that are critical to its mission and must operate continuously. A network provider, for example, loses both revenue and customer goodwill if its switches or service controllers are temporarily unavailable. Because of the need for continuous operation, planned downtime is hard to schedule, and unplanned downtime can have cataclysmic effects. When a new service or security threat makes changes unavoidable, the changes are designed to minimize downtime, reducing system maintenance to repetitious patching. Such patching breaks the program's data abstractions and encapsulation, reduces modularity and increases coupling. At the same time, ignoring needed maintenance and development leads to an outdated system, one that eventually will become an obstacle to organizational development.

Complicating the situation is that, as network computing has transformed many industries, continuous change has become just as important as continuous operation. Rapid introduction of new functionality and dynamic adaptation to volatile needs is essential. Systems, particularly telecommunications systems, must be customized on a very fine time scale, either due to user demand or to load and usage fluctuations.

An effective way to allow both continuous operation and continuous change is through dynamic code updates. New code is added to a running program *without halting the program*, thus introducing new functionality but avoiding downtime. The idea of adding new code to a running program dates back to the earliest electronic computers, and dynamic linking [1,2,3,4,5,6,7,8] is now available for nearly all operating systems and programming languages. However, since languages such as C++ do not directly support the creation of dynamically loadable modules, preserving program-level abstractions across the dynamic-linking interface is difficult. In particular, current dynamic linkers break the type safety of C++,[2] since they are oriented towards functions rather than types and classes.

Recently, new environments and languages have been designed to dynamically download and execute programs [9,10]. In particular, Java [9,11] has become extremely popular and is in widespread use. However, although a Java *class loader* can lazily load the classes that make up an application, it has no knowledge of class versions and can only load each class once. In addition, the relatively poor performance of Java makes it impractical for low-level applications.

Here we propose *dynamic classes*. Dynamic classes allow new functionality to be introduced into an exe-

[1] This work was done at AT&T Labs – Research.

[2] We recognize that C++ is not "type secure" since the programmer can break any type abstraction through explicit casts. Current dynamic linkers, however, break type abstractions even if the programmer does *not* perform explicit casting.

cuting C++ [12] program without sacrificing type safety, performance or the object-oriented paradigm. Replacing an entire class, rather than individual functions, honors the semantic integrity of the program and minimizes interference between the update and the ongoing computation. Although comparable techniques exist in interpreted languages and agent-based environments, our objective is to use these mechanisms in telecommunication (and other) systems where performance is the primary concern. We outline a C++ proxy-based implementation that requires only existing features of the language and is usable with any complete C++ compiler.

Section 2 presents related work. Section 3 introduces dynamic classes. Section 4 describes our proxy-based implementation. Section 5 examines system performance. Section 6 presents several applications in which we are using dynamic classes. Sections 7 and 8 consider future work and our overall conclusions. Finally, the appendix contains psuedocode for our proxy class. The psuedocode is discussed in Section 3.

2. Related work

The idea of adding new code to a running program dates back to the earliest electronic computers, but perhaps the first structured approach can be found in the rudimentary dynamic linking of the Multics system [1]. Since then some form of dynamic linking has found its way into a wide range of programming languages, distributed-computing environments and even operating-system kernels. Since we are targeting the C++ language, we first consider various ways to add code to an executing C or C++ program, and then examine a few approaches that have been used in other languages and environments.

2.1 C and C++

Incremental linking, or runtime linking of code that was available to the compile-time linker, achieves lazy loading of the code and avoids resource allocation for code segments that are never used [13,14]. Conceptually, however, incremental linking is identical to traditional compile-time linking.

Dynamic linking, or runtime linking of code that was not available to the compile-time linker, truly supports the introduction of new functionality into a running program [2,3,4,5,6,7]. However, the C++ language does not directly support the creation of dynamically loadable modules, making it difficult to preserve program-level abstractions across the dynamic interface. In particular, current dynamic linkers break the type safety of C++, since they are oriented towards functions rather than classes. For example, to create an instance of a previously *unknown* derived class, the program must search a shared library for the desired constructor via a tedious C interface (e.g., `dlfind` on many systems), and then *cast* the resulting function pointer to the constructor type. In addition, some dynamic linkers do not allow a previously loaded code module to be replaced later unless every call into the module is made via the same tedious C interface. Even when the dynamic linker allows the *relinking* described in [15], there is no support for replacing a module that is in use or having multiple versions of a module coexist within the same program.

Dorward et al. [8] extend dynamic linking so that it preserves the type safety of C++ and works at the class level. Their solution allows a *new* derived class of a *known* base class to be dynamically loaded into a program. First, the shared library that contains the implementation of the new class is dynamically linked into the program. Then, a standard factory mechanism [16] is used to call into the library and create an instance of the new class (a preprocessor automatically generates the necessary factory routines). The instance is cast to the type of the base class and can be used wherever an instance of the base class is expected. Since all calls to the base class are type checked statically, and the base class constrains the derived class to have the same function signatures, type safety is preserved even though the program is actually invoking the operations of the derived class. Our implementation uses much the same mechanism to achieve type safety, but it extends Dorward's implementation by (1) allowing the replacement of a previously loaded class with a new version and (2) allowing multiple versions of a class to coexist within the same program. In other words, our implementation adds versioning.

The techniques of Hamilton and Radia [17] and Goldstein and Sloan [18] do allow multiple versions of a class to coexist within the same program. In the Hamilton and Radia approach, the program must be recompiled to take advantage of a new version (and hence stopped and restarted). Goldstein and Sloan, on the other hand, allow the new version to be dynamically added to the running program, but their solution is meant for distributed systems in which programs communicate by passing objects to each other. Since the libraries that the programs use might be upgraded at different times, a program might receive an object for which it does not contain the corresponding library version. When this happens, the program dynamically loads the appropriate library version and directs all accesses to the received object into this library. Their solution is intended to be completely transparent and does not provide application-level control over the active version (although it could be extended to provide

such control). In addition, their solution requires non-trivial compiler support and does not allow an instance of an old class version to be passed to code that was compiled against a newer class version.

2.2 Other languages and environments

Java is of particular interest due to its widespread popularity and availability. Java is an object-oriented language that is syntactically similar to C++ [9]. A Java program is made up of one or more classes. Rather than load each class into the Java virtual machine at program startup, a *class loader* dynamically loads each class at the time of first reference. Although the built-in class loader has no knowledge of class versions and will only load each class once, it would be possible to write a custom class loader that took versioning into account. This custom class loader might need help from (1) Java *interfaces* to cleanly separate interface and implementation, (2) a preprocessor to enforce a version-naming scheme, and (3) some proxy-like class. Java, however, simply does not provide sufficient performance for the low-level applications of interest. It will be worthwhile to reconsider Java once effective just-in-time compilation brings its performance closer to that of C++. If Java becomes an attractive choice for our target applications, the same dynamic-class mechanism presented in this paper could be reimplemented in Java.

The Limbo language, which is part of the Inferno system from Lucent technologies, is intended for the same type of distributed applications as Java [10]. A Limbo program consists of one or more modules. Each module consists of a public interface and a private implementation. The public interface can include functions, variables, data types and constants. These modules can be dynamically loaded, unloaded and reloaded at runtime. Although versioning support would need to be added at a higher level, this dynamic loading capability would play a large role in a dynamic-class implementation for Limbo. Like Java, however, Limbo is not in widespread use for the applications in question and does not yet provide the desired performance.

Many other languages - such as Eiffel, Lisp, Perl, Python [19], Scheme, SmallTalk [20], Standard ML and Tcl [21] - support some form of dynamic linking or loading. Depending on the language, the dynamic update can be as small as a single procedure, a single class or a single compilation unit. Unfortunately, many of these languages are interpreted and are too inefficient for performance-critical systems. None of them directly provide the necessary versioning support. Most importantly, none of them are in widespread use in our application environment.

Most agent-based environments – which include mobile-code systems [22,23], cooperative processes [24], and intelligent interfaces [25] - allow the dynamic introduction and removal of individual process entities or agents. There is little support, however, for replacing an existing agent while preserving ongoing agent conversations. In addition, many agent systems use interpreted languages that are not efficient enough for low-level processing. Finally, due to the communication overhead in current agent systems, an application is often implemented as a few large agents rather than many small agents. These large agents are too coarse of a replacement unit for many applications.[3]

Many distributed programming systems such as CORBA [26] and Argus [27] also allow the dynamic introduction and removal of individual processing entities. In CORBA this entity is a single process. In Argus this entity is a collection of objects and processes called a guardian. Both systems provide limited support for replacement. As in the agent case, however, a process or guardian is too coarse a replacement unit for many applications, particularly since a guardian is unavailable during the replacement process (i.e, all ongoing conversations are blocked). Bloom, however, does make the notable contribution of analyzing when it is safe to replace one implementation of a guardian with another [27], an analysis that could be applied directly to dynamic C++ classes.

Finally, Microsoft's component object model (COM) [28] allows the programmer to define components, which have an interface and a separate implementation. If the implementation changes, existing components will continue with the old implementation, while new components can use the new implementation. This approach is quite close to the approach that we use for our dynamic classes. In addition, COM is efficient enough that components can be used at the "class" level. On the other hand, COM has many additional features that we do not need; it has a longer learning-curve than our simple proxy; and it is not widely used in non-Windows applications. Like Java, however, COM will be worth revisiting as it evolves.

3. Dynamic classes

A *dynamic class* is a class whose implementation can be dynamically changed during program execution, allowing the introduction of new functionality at the class level. Of course, the main program must be able to communicate with (invoke the methods of) the new im-

[3] As we will see later, however, our dynamic classes can be used to *implement* agent replacement (in systems where agent replacement is sufficient).

plementation. Thus each implementation must have an interface that is known to the main program at compile time. Each new implementation either updates an existing class (a new version) or introduces a new class (a new type) that uses the known interface.

3.1 Version Update Semantics

A basic problem when updating an existing dynamic class is what to do with existing objects. There are at least three approaches as shown in Figure 1. One approach (a) is to raise a "barrier," blocking object creation until all existing objects of older versions have expired. Then the new version takes over and object creation resumes. This approach is conceptually equivalent to halting, modifying and restarting the system. Another approach (b) is to *recreate* all existing objects using the new version. This approach retains a crucial property of raising a barrier, namely that at any time all objects of a particular class are of the same version. On the other hand, since different class versions can have different internal data structures, copying each object's state requires an understanding of the object's *semantics*.

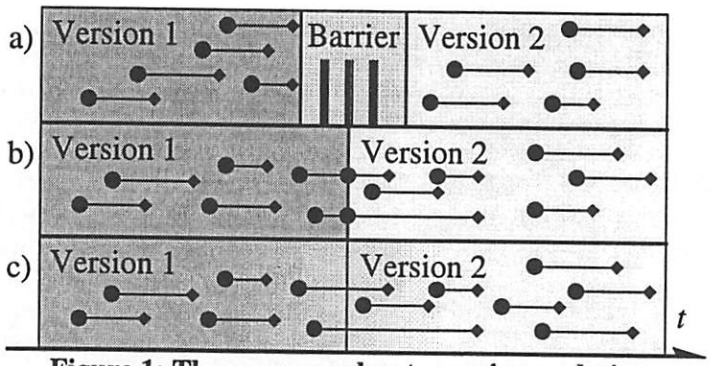


Figure 1: Three approaches to version updating

The last approach (c) is not to take any action at all. All new objects are created with the new version, and existing objects continue with their current versions. Once the existing objects finish their tasks and are destroyed, only the new version will be in use. We adopt this last solution since it is the most basic, is sufficient in all the cases that we have considered, and can be implemented efficiently. However, we also include a method that an object can use to determine if its version is the most recent version, allowing the programmer to explicitly migrate objects of a particular class. The programmer would need to write state-capture and restoration routines for each class version. These routines would produce and accept version-independent representations of an object's state.

3.2 The Interface Semantics

To allow "hot" updates, an interface monitor screens every message that passes through the dynamic class interfaces. This monitor is conceptually a class proxy. For each dynamic class, the monitor maintains a map that associates the class name with the current implementation version (and its location on external storage). A dynamic class is *invalid* if the running program does not contain any implementation version for that class. All dynamic classes start out as invalid. When a message is sent to an invalid class, the monitor locates and loads the current implementation version, updating the internal map as needed. The monitor then passes the message on to the newly loaded version.

The interface monitor provides three methods to manipulate the current version: *activate*, *invalidate* and *activate-and-invalidate*. The *activate* method registers a new version as the current version. Objects of older versions remain in existence. Once all old objects have expired normally, the older versions are removed from the system. The *activate-and-invalidate* method registers a new version as the current version but also invalidates (destroys) all objects of older versions. *Conversations*[4] in which these objects were engaged are broken. The surviving participants must recover and retry. The *invalidate* method invalidates (destroys) all objects of a given version. Invalidating the current version is an error. With these semantics, the three methods maintain the invariant that each dynamic class has a unique current version.

4. Implementation

We had three design goals for our C++ implementation of dynamic classes: (1) efficiently find and invoke the correct version of each method, (2) hide the dynamic class mechanism as much as possible from the end programmer, and (3) use only standard C++ features. The implementation should not require special preprocessor or compiler support, nor depend on *compiler-specific* details. These design goals led us to proxy classes. In this section, we first present our proxy-class implementation, and then discuss the tradeoffs involved.

[4] We use the term *conversation* for a sequence of message exchanges (or method invocations) between two objects. In our implementation, if one of the objects is dynamic and is invalidated, the other object will receive an exception when it attempts to invoke the invalidated object's methods. This other object must then perform some application-specific recovery, such as constructing a new object of the same class as the invalidated one. Clearly the programmer should think carefully before using invalidation.

The core of our implementation is a generic template class. The template, which is shown in the appendix, uses only standard C++ features and can be compiled with any complete C++ compiler. The template serves as a proxy (or *smart pointer*) for each dynamic class. As with any proxy, a program creates a dynamic class instance by creating a proxy instance instead.

Each dynamic class is written as two separate parts: (1) an abstract interface class that is known to the program at compile time and (2) one or more implementation classes that inherit from the interface class. There is one implementation class for each version of the dynamic class. The abstract interface class specifies the public operations that remain constant across all versions of the dynamic class. These operations are defined as pure virtual functions so that each derived implementation class is forced to provide them. In addition, although each implementation class can have any additional methods and data members that it needs to perform its task, only the operations defined in the interface class can be called from other program modules. This only makes sense since otherwise a program module might become dependent on a particular version of the dynamic class. Each implementation class is compiled into a separate shared library.

To use a dynamic class, the template is instantiated on the interface class. At run-time, the template locates the shared library that contains the most recent implementation class and loads this library into the program's address space. The template calls into the library to create an instance of the implementation class, and casts the instance to the type of the interface class. Finally, the public interface operations are accessed through the template using standard pointer redirection.

The template also provides static methods that implement *active*, *invalidate* and *activate-and-invalidate*. Most software systems will provide an external interface through which an administrator, developer or automated management tool can invoke these methods.

As an example, consider a dynamic class whose job is to receive packets sent across a network connection. For simplicity, the dynamic class interface provides only a single operation.

```
class Receiver {
  public:
    virtual Packet receivePacket (void) = 0;
}
```

The programmer might write two implementation classes, the normal production version and later, after the discovery of an unexpected problem, a debugging version that contains new debugging code.

```
class ReceiverImp: public Receiver {
  public:
    Packet receivePacket (void) {…}
}

class ReceiverDebuggingImp: public Receiver {
  public:
    Packet receivePacket (void) {
      logDebuggingInfo(); …
    }
}
```

Each of these two implementation classes is compiled into its own shared library, say *imp.so* and *debugimp.so* respectively. Using the dynamic class is now straightforward (*dynamic* is the name of our proxy template as shown in the appendix).

```
// normal program operation - create and use
// normal packet receivers
dynamic<Receiver>::activate ("imp.so");
dynamic<Receiver> receiver;
Packet packet = receiver.receivePacket();
…
// switch to debugging mode in response to
// some external event (library name would
// be included in the external event)
dynamic<Receiver>::activate ("debugimp.so");
// now all new packet receivers will contain
// the debugging code
dynamic<Receiver> otherReceiver;
…
```

Thus new debugging functionality is introduced without stopping the running program. In addition, once the bug is identified, the developer can create a third version of Receiver, one that contains the necessary fix. The fixed version can then be activated, again without stopping the running programming. Of course, we do not intend to suggest that all bugs can be fixed without stopping the program, but at least some bugs can be. For example, our Receiver might simply be mistranslating a particular kind of packet.

4.1 Implementation details

Construction and invocation. Since there can be multiple versions of a dynamic class within a program, there can be multiple implementations of each method within the program's address space. The correct implementation must be called when a method is invoked on a particular object. The problem is to have a method invocation that is version-dependent, but that can be resolved at compile time by a C++ compiler that (1) is oblivious to versioning and (2) has access to only to the interface class. The solution is a two-level indirect method resolution at runtime. The first level is the version-dependent mapping, which we implement ourselves inside the dynamic class proxy; the second level is the method mapping within a version, which we can achieve with standard C++ virtual methods (and the associated vtables).

In our approach, the version of each object remains unchanged for the lifetime of the object. Therefore, the version mapping can be resolved during object creation

and stored in the instance of the proxy. In fact, once the object is created, the needed mapping is just a single pointer to the new object (namely the `object` pointer that appears in the template definition in the appendix).

The method mapping is achieved in C++ by defining all methods in the interface class as virtual. C++ adds a vtable to each derived implementation class, and resolves all method calls through the vtables [12]. Figure 2 illustrates the two-level mapping. The proxy contains a pointer to the object; the object contains a pointer to the vtable; and the vtable contains pointers to the methods of the object's implementation version.

Since the vtable is not used when invoking a constructor, the problem of how to actually construct the object remains. Since each implementation class has (1) its own constructors and (2) possibly a different size, normal C++ constructor syntax can *not* be used. Instead we use the standard factory pattern and require each class version to provide a static method, *createInstance,* which the proxy calls instead of the constructor. A side effect of this approach is that each dynamic class essentially has only one constructor, namely the one that *createInstance* chooses to invoke. Additional initialization must be done through other methods.

External map. Each version of a dynamic class is compiled into its own shared library. At runtime, the *activate* and *activate_and_invalidate* methods must be able to locate the correct library given some symbolic name for the desired class version. There are several ways to accomplish such a mapping, but we found that the easiest approach for the end programmer was to simply use the library name (without its path) *as the symbolic name.* The two methods then search all known library directories for the given library.[5] Of course, the library name does not need to be known to the program at compile time; it can be passed to the program at runtime during the version-update process.

We have also extended our proxy so that the symbolic name can be an arbitrary URL, both to support network applications and to use Web technology for the storage of dynamic class libraries. If the URL refers to a library on the local machine, the library is immediately loaded into the program's address space. If the URL refers to a library on a remote machine, the library is first downloaded onto the local machine. A further description of our network support is beyond the scope of this paper. Relevant issues include caching the libraries on the local machine, handling different machine architectures,

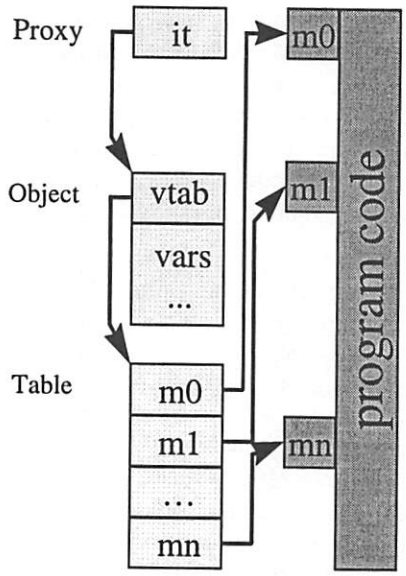


Figure 2: Method invocation

checking the security credentials of the downloader, and imposing a consistent, meaningful naming scheme.

Template versions. The dynamic class system provides three versions of the proxy template. One version has restricted functionality but high performance; another has full functionality but slightly lower performance; and the third falls between the first two on both functionality and performance. The full-functionality version allows different dynamic classes to share the same interface class, and therefore allows the introduction of *new* dynamic classes into the system. For example, a programmer could write a *NetworkConnection* interface class, and then write the dynamic classes *TcpipConnection*, *RpcConnection*, etc., which all use *NetworkConnection* as their interface. *TcpipConnection*, *RpcConnection*, etc., could all have multiple versions. Thus this version of the template provides two "mappings". For each dynamic class, there is a list of the class versions that are currently present inside the program, and for each interface class, there is a table that associates each dynamic class name with the correct version list. This full-functionality version of the template is shown in the appendix. Note that *activate*, *invalidate* and *activate_and_invalidate* take two parameters, the name of the dynamic class and the name of the library that contains the desired version of that class. Note also that *activate* and *activate_and_invalidate* return a unique handle for the dynamic class, which is passed to the template constructor to identify the desired dynamic class. Handles make the constructor much faster, since looking up a handle is much more efficient than looking up a class name.

The medium-functionality template requires each dy-

[5] How library directories are specified is platform dependent. On most Unix machines, library directories are listed in the environment variable LD_LIBRARY_PATH.

namic class to have its own interface class (and thus supports version updates only). The template still maintains a version list for each dynamic class, but eliminates the class-name to version-list mapping, avoiding (1) the string lookup in the *activate*, *invalidate,* and *activate_and_invalidate* methods and (2) the handle lookup in the constructor. In this template, the three methods take only the library name as a parameter, and the constructor takes no arguments. The class name, which is needed to invoke the constructor, is obtained from a special function in the shared library (i.e., a static function that returns the class name as a `const char *`). The medium-functionality template is the most commonly used template. Its performance is discussed in Section 5.

Finally, the low-functionality template does not support the *invalidate* and *activate_and_invalidate* methods. New versions can be introduced, but old versions can not be invalidated. Since each version remains valid as long as objects of that version exist, the redirection operator does not need to check for invalid versions. More importantly, the redirection operator no longer throws exceptions, which allows C++ compilers to inline the operator and eliminate one function call. Together these two things make method invocation much faster as will be seen in Section 5. All three versions of the template can be freely used in the same program.

4.2 Implementation tradeoffs

We chose the proxy approach since the ability to use standard C++ development environments was a primary design goal. If we had been willing to sacrifice this goal, we would have had more implementation choices, most notably a custom preprocessor. A sufficiently complex preprocessor could (1) make dynamic classes look more like normal C++ classes (i.e., multiple constructors per dynamic class and method access via selection (.) rather than just deference (->)), and (2) provide better performance. A careful analysis of the tradeoffs, however, indicates that the preprocessor advantages are not as great as they first appear, and do not justify the additional complexity and development time.

Performance. The proxy approach involves two overheads. First, the proxy constructor must find and invoke the correct factory method. The factory method then invokes the real constructor. Second, all method accesses go through the proxy's dereference operator, which involves an extra function call (in the absence of inlining) and a Boolean comparison to verify that the class version has not been invalidated. Although custom preprocessor support could not eliminate this entire overhead (i.e., there must be a mapping from an object to its version), it would eliminate at least some levels of indirection. Here, however, it is worth considering that a typical dynamic class will not be the finest-grained class in the system. It is unlikely that a programmer will need a dynamic *Point* class if a *Point* is simply a coordinate in two-dimensional space. It is more likely that the programmer will need a dynamic *Renderer* class that draws some figure given a set of *Points*. The methods in *Renderer* would do significant processing, and the extra time needed to just invoke the methods would be insignificant. Our expectation is that most (if not all) dynamic classes will fall into the same category as *Renderer* and perform nontrivial processing in most of their methods. All of the application work so far confirms this view. For this reason, we feel that the performance issues are minor, and do not justify any custom preprocessor or compiler support. If our expectations are wrong, however, and programmers start to make featherweight classes dynamic, we will need to re-examine our proxy approach. We will say more about performance in Section 5.

One technique that does not involve any custom support is to re-map the vtable associated with a particular dynamic class version in response to certain events. For example, the *invalidate* operation could make every entry in the vtable point to a dummy method that throws an exception. Then the dereference operator would not need to check if the class version were still valid. The proxy is still necessary, however, since the proxy provides additional functionality behind the scenes. For example, the proxy maintains a count of all objects of a particular version, so that the version code can be removed from the program's address space as soon as those objects have been destroyed.[6] Thus, given that (1) the proxy is still necessary, (2) working with vtables does introduce a few compiler dependencies, and (3) the performance penalties without re-mapping are small, we believe that vtable re-mapping will not provide sufficient benefits to be worthwhile.

Abstraction hiding. The proxy approach means that the program (and hence the programmer) must explicitly know about dynamic classes at some level. In addition, the proxy approach prevents the usage of some standard C++ constructs. Most notably, a dynamic class can have only a single constructor, and all method access must take place through the dereference operator.[7] Finally, our proxy implementation allows a programmer to obtain a direct reference to the embedded object (through a perhaps atypical use of the dereference operator). The programmer could then access the

[6] Removal occurs only if the version is no longer the active version.

[7] Passing a *reference* to a dynamic class is fine, but method access still involves the dereference operator.

object without going through the proxy class. Preprocessor and compiler support could address all of these problems. However, it is a simple C++ programming task to write a wrapper class that contains an embedded instance of our dynamic-class proxy. This wrapper class could (1) hide the existence of the proxy from the rest of the program, (2) provide multiple constructors, (3) support normal C++ access syntax, and (4) prevent the client code from obtaining any direct reference to the actual versioned object. For example, here is a wrapper for our Receiver class (assuming that the interface has been extended with an *initialize* method).

```
class ReceiverWrapper {
  private:
    dynamic<Receiver> *receiver;
  public:
    ReceiverWrapper (void) {
      receiver = new dynamic<Receiver>;
    }
    ReceiveWrapper (int packetType) {
      receiver = new dynamic<Receiver>;
      receiver -> initialize (packetType);
    }
    Packet receivePacket (void) {
      return (receiver -> receivePacket());
    }
}
```

The wrapper class looks like a normal class to the rest of the program. The only exception is the program module that accepts versioning instructions from external sources (and passes these instructions on to the appropriate proxies). Given the ease with which these wrapper classes can be written, we again felt that custom preprocessor or compiler support was not justified. Of course, the fact that a wrapper class can be written does not mean that it will be written. The programmer is free to bypass our proxy methods, breaking the dynamic-class abstraction. It is hard to imagine how the programmer could do this accidentally, however, and no programmer has done it accidentally so far. Thus we are content to provide a flexible mechanism without *enforcing* all usage requirements.

In a similar vein, existing programs cannot use our dynamic classes without modification. In a large software system, however, it is likely that these modifications would be confined to a particular subsystem, already hidden behind an appropriate class. In addition, it is difficult to imagine that an existing program could use *any* dynamic class implementation without modification. Unless we have a preprocessor or compiler that essentially makes all classes dynamic, we must at least add some syntactic markup and then recompile.

Finally, the behavior of static methods in a dynamic-class interface is currently undefined. This is mainly an implementation detail. The static methods must be compiled only into the main program, not into any of the shared libraries. Then all static method invocations will be directed to the same code (which is what we want since the methods will only make sense if they perform version-independent processing). Compiling the static methods only into the main program does not require any special support, although some preprocessor support would help the programmer avoid mistakes.

Inheritance. The proxy approach complicates inheritance in several ways. First, the proxy approach demands a clean separation between the interface and implementation of a class, simply because the interface and implementation *must be* separate classes. Unfortunately, such a clean separation is not seen in many existing C++ programs. On the other hand, any dynamic class implementation will require the same separation, since a dynamic class must have a version-independent interface. Otherwise client code would quickly become dependent on particular versions.

Second, the proxy restricts how dynamic classes are inherited. In general, interface classes inherit from other interface classes; an implementation class inherits from other implementation classes; and a *normal* class that wants to extend the functionality of a given dynamic class *contains* an instance of the appropriate proxy. Preprocessor or compiler support could certainly relax this strict separation. In the same light as some of the other issues, however, it is questionable whether such a relaxation should be allowed. Imagine, for example, if a dynamic class inherits from a particular version of some other dynamic class. Then, whenever the system constructs an instance of the subclass, it must identify and use the correct version of the superclass (and of the superclass of the superclass). Although there are programmers who could keep the resulting dependencies straight, the same inheritance effect can be achieved much more cleanly and easily with separate interface and implementation hierarchies (and without complex compiler or preprocessor support).

Third, our proxy does have undesirable consequences for interface polymorphism. Even if one *interface* inherits from another, the *proxy class* instantiated on the derived interface is *not* related to the *proxy class* instantiated on the base interface. Therefore, contrary to the intent of the interface inheritance, the proxy of the derived interface cannot be used where a proxy of the base interface is expected. Our solution is to provide a template function that performs an explicit cast; the template function succeeds (at compile time) only if the respective interfaces are related through inheritance. Although our solution is sufficient, simple preprocessor support would be useful here.

Finally, the full implementation of any superclasses must be compiled into the same library as the dynamic

class version to ensure that all superclass references are resolved correctly (at compile time). Although this means that the superclass code might appear in multiple libraries, it simplifies our implementation significantly and involves minimal extra work for the programmer. Again some preprocessor support would be useful here.

Summary. In contrast with a preprocessor- or compiler-based approach, our proxy solution is much simpler, but has lower performance, does not fully hide the dynamic class abstraction, and restricts inheritance. However, the performance penalty is almost always small relative to the processing that the dynamic classes are performing; the proxy can be hidden completely with a straightforward wrapper class; and *any* dynamic-class implementation will likely restrict inheritance so that inheritance remains understandable.

5. Performance Evaluation

5.1 Time Complexity

We ran two tests, one measuring the time to construct and destroy an object (the class had no data elements and its constructor had no arguments), and the other measuring the time to invoke an object method (the method had no arguments and no return value). Each test involved three cases: (1) a standard stack-allocated class that does not have virtual methods, (2) a standard heap-allocated C++ class that has virtual methods and (3) a dynamic class created through our medium-functionality template. We considered both case (1) and (2) since a dynamic class always involves virtual functions and heap allocation. Each test was compiled with the SGI C++ compiler and was performed ten million times per run for ten runs on an SGI Indy.

When the constructor is empty, the construction overhead of dynamic classes is 80% versus the heap-allocated class, and 1091% versus the stack-allocated class. When the constructor zeroes out a 128-byte block of static memory, the overheads drop to 14% and 160% respectively. In a multi-threaded environment, the time to acquire a lock (to prevent corruption of the version lists) dominates the construction process, and the overheads increase to 650%/119% (empty/non-empty constructor) versus the heap-allocated class, and 4872%/160% versus the stack-allocated class.

When the method is empty, the invocation overhead of dynamic classes is 240% versus the heap-allocated class, and 611% versus the stack-allocated class. When the method defines three local variables, increments the value of these variables by one, and performs three integer comparisons (that evaluate to false), the overheads drop to 110% and 175% respectively. In addition, if *invalidation* is not required, the low-functionality template can be used instead of the medium-functionality template. The low-functionality template has much better performance, since its redirection operator does not throw an exception and can be inlined by the compiler.[8] With this template, the overheads are 39%/18% (empty/non-empty method) versus the heap-allocated class, and 191%/55% versus the stack-allocated class.

The performance penalty of making an existing C++ class dynamic is high if its methods are (nearly) empty. The penalty is quite low, however, if its methods do any nontrivial processing. Thus our dynamic-class implementation is not appropriate for a low-level class such as *Point*, since *Point* is (1) computationally trivial (each method does little more than a single assignment) and (2) small enough to allocate on the stack. In addition, if *Points* are used throughout the system, they will be created and destroyed constantly, leading to a severe performance penalty since the proxy constructor takes much longer than the simple *Point* constructor. However, our dynamic-class implementation is appropriate for most higher-level classes such as the *Renderer*, especially if the class objects (1) are allocated on the heap anyway, (2) are created and destroyed infrequently, or (3) do nontrivial processing in their constructors and other methods. As discussed above, we expect that dynamic classes will be used only with high-level classes anyway. Our implementation provides excellent performance for these classes. In all of the applications that we have implemented at AT&T, for example, the dynamic-class methods do far more processing than the test cases presented in this section, and the overheads are insignificant.

5.2 Space Complexity

The space requirements of dynamic classes are low. For each *class*, the proxy maintains a version list, a pointer to the active version (so that it does not have to search the list during construction), and a synchronization lock (in a multi-threaded environment only). The version list has one entry per version. Each entry contains the associated class and library names, a flag that indicates whether the version is the active version, and a count of the number of objects of the version. For each *object*, the proxy maintains two pointers, one to the actual object and one to the object's version information inside the version list. In addition, since a dynamic class always has virtual functions, the actual implementation object has a pointer to the appropriate vtable.

Thus the *per-object* space overhead of dynamic classes is three pointers (including the vtable pointer), and dy-

[8] Most C++ compilers will not inline a method that can throw an exception.

namic classes can be used only if this overhead is acceptable. For the *Point* class, the overhead is probably unacceptable, since the three pointers might take up more space than the *Point*'s actual data. For the *Renderer* class, the overhead is probably acceptable. In the applications that we discuss below, the three pointers are less than 10% of the total object size.

6. Applications

Our main motivation for this work is network-control and service-management applications that (1) demand the high performance of C++ but (2) must operate continuously. We have used dynamic classes in three such applications at AT&T, namely mobile agents, control-on-demand and connection management. The *mobile-agent* application is a building block for the control-on-demand application, but also demonstrates the viability of native code in a heterogeneous environment. The *control-on-demand* application uses agents to inject application-specific control policies into a router. Finally, the *connection-management* application uses dynamic classes to inject handlers for new connection types into a running connection manager.

6.1 Mobile Agents

A mobile agent is an executing program that can migrate *at times of its own choosing* from machine to machine in a heterogeneous network [22]. Mobile agents are attracting growing attention as a means to easily realize complex, distributed applications, and are being used at AT&T in several prototype network-management applications. We have implemented an efficient mobile-agent system on top of our dynamic-class mechanism. Each agent is a version of the same dynamic class. The interface class defines the operations that are common to all agents, most notably *migrate*, *captureState*, *restoreState* and *run*. Unlike most dynamic classes, the interface class *implements* the *migrate* operation. The agent implements the other three. Like all dynamic classes, each agent is compiled into its own shared library.

A mobile agent starts executing when a bootstrap program loads the agent (via the dynmic-class mechanism) and calls its *run* method. The *run* method performs the agent's task. If the *run* method decides that the agent should migrate to another machine, it calls the *migrate* method. The *migrate* method calls the *captureState* method to package up the agent's current state, and then transmits the state image and the *URL* of the agent's shared library to a server on the target machine. The server downloads the shared library, loads the agent (again via the dynamic-class mechanism), calls the *restoreState* method to restore the agent's state, and finally calls the *run* method. The *run* method continues with the agent's task, checking the agent's current state to decide what to do next.[9]

6.2 Control on demand

Control-on-demand [29] is flow-oriented active networking. Applications inject customized control policies for each flow into the network routers. These policies exploit strategic positioning, local network knowledge and application semantics to improve the performance or perceived quality of the flow. They may act both in the control plane and the data plane. The former supports connectivity control, such as floor management for a teleconference or advanced group management for a multicast. Applications of the latter include (1) stream thinning at a branch point in a variegated multicast, (2) discarding less important packets during congestion (e.g., discarding B and P frames to protect I frames in an MPEG stream), or (3) monitoring packet loss and retransmitting lost packets from inside the network.

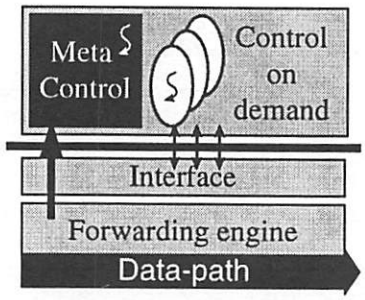


Figure 3: A node supporting control-on-demand

Figure 3 shows a control-on-demand network node. The two major parts are a forwarding engine (below) and controllers (above) separated by an opaque interface. The meta-controller accepts, installs and runs new controllers on demand, using dynamic classes in the same way that they were used for mobile agents.

6.3 Service/connection management

To experiment with dynamic classes in service-management applications, we have built a prototype connection manager. Before transmitting, an application issues a connection request to the connection manager. The manager creates a connection instance, performing admission control and other "book-keeping" in the process. On disconnect, the instance is destroyed, and resources allocated to the connection released. While the connection is active, the instance is responsible for ensuring service quality.

---

[9] Since different machine architectures cannot use the same shared library, the agent must be pre-compiled for all machine architectures on which it might find itself. Each machine inserts its architecture "name" into the URL before downloading the library. This naming scheme, as well as other issues such as security and fault-tolerance, is beyond the scope of this paper.

The static part of the connection manager (the main program) recognizes only a general interface to connection objects. Type-specific implementations of this interface are introduced dynamically (using the full-functionality version of dynamic classes). Figure 4 depicts a hierarchy of connection types that is dynamically constructed during the execution of the connection manager. The manager is an event processor that handles events of the form: {*connection-type name, data (initial state), code reference (URL)*}. If the connection type is not known, the code is introduced as a new type. If the type is known, but the code reference has changed, the code is introduced as a new implementation version. In either case, the new code is retrieved and installed using the dynamic-class mechanism.

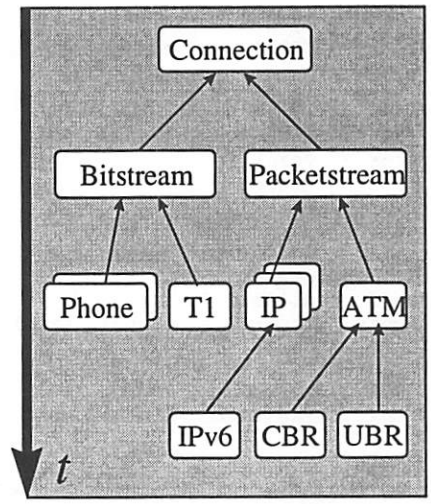


Figure 4: A hierarchy of connection types

7. Future work

The first area of future work is to add security mechanisms to our dynamic-class implementation so that a program can verify the origin of the dynamic classes that it is instructed to load.

A second area of future work is to handle the case where several dynamic classes must undergo a version update as an atomic unit. Potential solutions include a simple transaction mechanism or a constraint definition language (i.e., new versions can be loaded at any time, but become active only when all constraints are met).

Finally, we are working with other groups at AT&T to identify additional applications for dynamic classes. We also plan to provide a dynamic-class implementation for Java. Although Java is unsuited for low-level control software, it can be used to implement higher-level components in telecommunications systems.

8. Conclusion

Dynamic classes provide powerful support for the maintenance and extension of mission-critical and other long-running applications. New implementations of a class can be dynamically added to and removed from a running program, eliminating the need to bring down the program when fixing bugs, enhancing performance, or extending functionality. The implementation discussed in this paper provides an easy-to-use dynamic-class library for the C++ language. The implementation preserves type safety and the class abstraction. It does not require special compiler support, and is efficient enough for use in low-level software.

We have already used dynamic classes in a mobile-agent application and as part of a larger programmable-network effort. In our experience, dynamic classes are efficient, easy to use and sufficient for most tasks.

9. Availability

Our dynamic-class implementation is available for Irix, Solaris and Windows 95/NT. It is known to compile under Irix with release 4.0 of the AT&T C++ compiler, under Solaris with release 4.2 of the Sun C++ compiler, and under Windows 95/NT with release 4.2 of the Microsoft C++ compiler. Porting to another platform is simple, as long as the platform has (1) a C++ compiler with exception and template support and (2) dynamic-linking facilities comparable to those of Irix. Interested readers should contact the second author.

10. Acknowledgments

Many thanks to the anonymous reviewers and our shepherd, Benjamin Zorn, for their excellent feedback, and to the AT&T programmers who have put our dynamic-class implementation through its paces.

Appendix

The following C++ code is a simplified version of the full-functionality template. As discussed in the paper, this template allows different dynamic classes to share the same interface class. Other templates support version updates only to achieve higher performance (by eliminating a table lookup). The template methods all throw C++ exceptions on error.

```
typedef list<DynamicVersion*> VersionList;
typedef int Handle;

template<class T> class dynamic {
    // the actual object, its version
    // information and its class handle
    T *object;
    DynamicVersion *dvPtr;
    Handle classHandle;
  public:
    // constructors, etc.
```

```
    dynamic(Handle handle);
    dynamic(const dynamic<T>& proxy);
    ~dynamic();
    dynamic<T>& operator= (
      const dynamic<T>& proxy);
    // smart pointer
    T *operator-> (void);
    // activate a version
    static Handle activate (
      const char *libraryName,
      const char *className = NULL);
    // invalidate a version
    static void invalidate (
      const char *libraryName, Handle handle);
    // activate and invalidate
    static Handle activate_and_invalidate (
      const char *libraryName,
      const char *className = NULL);
  private:
    // static data (shared by all versions and
    // classes implementing the interface <T>)
    // 1. map: handles to version list
    static VersionList **versionMap;
    // 2. map: handles to active version
    static DynamicVersion **activeMap;
    // ... more ...
};
```

References

[1] F. J. Corbato and V. A. Vyssotsky, "Introduction and Overview of the Multics System," Proceedings of the AFIPS Fall Join Computer Conference, 1965, pp. 185-196.

[2] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. "Shared Libraries in SunOS.," *Proceedings of the USENIX Summer Conference*, 1987, pp. 375-390.

[3] James Kempf and Peter B. Kessler, "Cross-Address Space Dynamic Linking," Technical Report TR-92-2, Sun Microsystems Laboratories, Inc., Mountain View, California, 1992.

[4] W. Wilson Ho and Ronald A Olsson. "An approach to genuine dynamic linking," Software-Practice And Experience, volume 21, number 4, April, 1991, pp. 375-390.

[5] Donn Seeley. Shared Libraries as Objects. USENIX Summer Conference Proceedings, 1990, pp. 25-37.

[6] March Sabaella. "Issues in Shared Library Design," USENIX Summer Conference Proceedings, 1990, pp. 11-23.

[7] Michael Franz. "Dynamic linking of software components," IEEE Computer, volume 30, number 3, March, 1997, pp. 74-81.

[8] Sean M. Dorward, Ravi Sethi and Jonathan E. Shopiro. "Adding New Code to a Running C++ Program," Proceedings of the USENIX C++ Conference, 1990, pages 279-292.

[9] "The Java Language: A White Paper," Sun Microsystems White Paper, Sun Microsystems, 1994.

[10] "Inferno: la Commedia Interattiva," Lucent Technologies White Paper, Lucent Technologies, Inc., 1997.

[11] Mary Campione and Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet*, Addison-Wesley, 1996.

[12] Bjarne Stroustrup. The C++ Programming Language (3rd Edition), Addison Wesley, 1997.

[13] J. J. Puttress and H. H. Goguen, "Incremental Loading of Subroutines at Runtime," Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986.

[14] R. W. Quong, "The Design and Implementation of an Incremental Linker," Technical Report CSL-TR-88-381, Computer Systems Laboratory, Stanford University, 1989.

[15] David Keppel and Stephen Russell. "Faster Dynamic Linking for SPARC V8 and System V.4," Technical Report 93-12-08, University of Washington, 1993.

[16] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[17] G. Hamilton and S. Radia. "Using Interface Inheritance to Address Problems in System Software Evolution," *Proceedings of the ACM Workshop on Interface Definition Languages*, 1994.

[18] Theodore C. Goldstein and Alan D. Sloane. "The Object Binary Interface – C++ Objects for Evolvable Shared Class Libraries," Technical Report TR-94-26, Sun Microsystems Laboratories, Inc., Mountain View, California, 1994.

[19] Mark Lutz. *Programming Python*. O'Reilly, 1996.

[20] A Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, Massachusetts.

[21] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[22] James E. White. "Telescript Technology: The Foundation for the Electronic Marketplace," General Magic White Paper, General Magic, Inc., 1994.

[23] Robert S. Gray. "Agent Tcl: A Flexible and Secure Mobile-Agent System," Mark Diekhans and Mark Roseman, editors, *Proceedings of the 4th Annual Tcl/Tk Workshop*, Monterey, California, July, 1996.

[24] Michael R. Genesereth and Steven P. Ketchpel. "Software Agents," Communications of the ACM, 37(7), July, 1994, pages 49-53.

[25] Yezdi Lashkari and Max Metral and Pattie Maes. "Collaborative Interface Agents," *Proceedings of AAAI '94*, 1994

[26] Jon Siegel, *CORBA: Fundamentals and Programming*, Wiley, 1996. ISBN 0471-12148-7.

[27] Toby Bloom. *Dynamic Module Replacement in a Distributed Pogramming System* Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

[28] "The component object model (COM): A technical overview," Microsoft White Paper, Microsoft, Inc., 1996.

[29] Gísli Hjálmtýsson and Samrat Bhattacharjee. "Control on Demand - Customizing Control for Each Application," AT&T Technical Memorandum, 1997.

Fast Consistency Checking for the Solaris File System

J. Kent Peacock, Ashvin Kamaraju, Sanjay Agrawal
{kent,akk,sanj}@eng.sun.com
Sun Microsystems Computer Company

ABSTRACT

Our Netra NFS group at Sun set out to solve the challenging problem of providing remote Network File System (NFS) service with high performance and availability. An NFS server must guarantee the permanence of changes to the file system before acknowledging an NFS request. Thus, the server's underlying local file system must perform update operations synchronously to stable storage with potentially high latency. Our solution to this problem involves using the Solaris Unix File System (UFS), derived from the Berkeley Fast File System (FFS), in conjunction with nonvolatile RAM (NVRAM) as fast stable storage. We evaluated the system using the LADDIS benchmark and as a result, developed a cacheing technique for block-mapping information that gave us a 23% increase in measured server throughput in our standard RAID-5 server configuration. With recent increases in disk capacity and RAID technology, file-system sizes have reached a point not imagined by the FFS designers, requiring an approach to checking file-system consistency that does not grow proportionately with file-system size. We examined several log-based solutions to providing fast crash recovery, but none could use the NVRAM effectively and meet our performance requirements. As an alternative, we developed an approach that uses UFS but maintains file-system working-set information, so that the consistency checker needs to examine only the active portions of a file system. This approach met our performance goals and also reduced file-system consistency-checking times to between 3% and 25% of those in the original UFS implementation.

1 Introduction

The goal of the Netra NFS project at Sun was to produce a dedicated NFS file server with performance and availability that would satisfy the increasing requirements of client–server networks.

Since the work was to be performed in the context of the Solaris Operating System, the natural file system to use was the native UFS file system. UFS was derived from the Berkeley UNIX Fast File System developed during the 1980s [McKusick84]. At that time, disks and file systems were small and slow relative to those found in today's computing systems, and conglomerations of drives were not in general use. Thus, UFS was not designed to handle gracefully file systems that can be hundreds of gigabytes.

One of the features of UFS is that a large volume of data is maintained in a set of caches in main memory, and is flushed back to disk in the background. This feature improves file-system performance significantly, though at the expense of possible loss of recently written data when the system crashes. In addition, it is necessary to perform a file-system consistency-checking operation on reboot after a crash in order to ensure reliable operation after the next mount of the file system. Without this consistency check, files and data could be lost or corrupted. The program that does this task, *fsck*(1M) [McKusick94], is required to find and correct inconsistencies; it does so by reading all the control information contained in the file system and correcting that information when it discovers errors. The amount of information is proportional to the size of the file system. Thus, as file systems have grown huge, the time to perform the consistency check has become unacceptably large.

A widely used solution to this problem has been the application of traditional database-logging techniques to allow fast recovery. The log contains information about the most recent transactions on the file-system control data, and so the log replay can restore the file system to a consistent state after a crash. This approach is much faster than *fsck* because only those files referenced in the log need to be processed.

Logging solutions are less desirable in the context of an NFS file server, however. To reduce disk writes, typical implementations batch many transactions into a single log write. This batching is especially effective when there are many operations that are asynchronous, since the originator of an asynchronous operation does not require that the transaction be completed before it continues. As noted, this behavior is typical for a local

UFS file system. However, when a UFS file system is accessed remotely through NFS, the NFS protocol definition requires that the transactions for most operations be complete before acknowledgement is returned to the client. With few or no asynchronous operations, the expected batching does not occur, and the log can become a significant bottleneck because each transaction generates a separate write to the log.

As an alternative approach, we considered creating a faster file-system checking program. At any given time, there is a limited set of files and auxiliary control information active on a file system. If it were possible to keep track of the active portions of the file system, the scope of the file-system check could be reduced dramatically to only the "working set" of the file system. The "fast fsck" provided with Netra NFS records the working set by adding a small amount of state information to the file system and performing "lightweight logging" of certain transactions in the file system, such as directory operations. The file system accomplishes this lightweight logging efficiently by adding transaction state information to a reserved portion of disk blocks containing file inodes. The mini-log can be thought of as a log that is distributed throughout the file system. The net result is a file-system checking program that takes a fraction of the time used by the original UFS *fsck* program.

2 Previous Work

The current Solaris UFS file system follows its predecessor, the Berkeley Fast File System (FFS) [McKusick84], in its approach to providing file-system robustness. It uses synchronous writes of metadata in a sequence such that the *fsck*(1M) program can restore the file system to a consistent state after a system crash. Without ancillary information, *fsck* must assume that all file-system metadata may be inconsistent, and must therefore scan all of the metadata to find any inconsistencies present. This scan takes unacceptable time in enormous file systems.

The most commonly used solutions to the problem of providing fast recovery are borrowed from database design — namely transaction logging (e.g., [Hagmann87, Chutani92]) or shadow paging (e.g., [Seltzer93, Hitz94]) of file-system metadata and, in some cases, of actual file data. The Veritas File System (VxFS), although not described in the literature, is prevalent in commercial UNIX systems; it uses transaction logging for file-system data and metadata. An interesting approach to avoiding the necessity of synchronous writes, called *soft updates*, is presented in [Ganger94]. This approach shows performance gains of a factor of between 2 and 15 on metadata-intensive benchmarks, and reduces the consistency-checking time from 5 to 7 minutes to 3 to 5 seconds, but only in a local file system environment where file I/O is asynchronous. In an NFS context [Sandberg85, NFS94], almost all remote file operations are synchronous with respect to the client. Thus, it is doubtful that soft updates would provide as great a performance benefit in an NFS environment, although the fast-reboot benefit would still be obtained.

Vahalia and colleagues [Vahalia95] dealt with the same problem that we faced. In fact, the first two sections of their paper give an excellent summary of what is required to provide good NFS server performance. Their approach uses metadata logging at the UFS level as a *redo-only* log that records only the new value of each modified object and can roll completed transactions only in the forward direction. They solve the problem of NFS requests requiring synchronous commitment to stable storage by batching writes to the log, which is set up as a separate disk device. The goal is to keep the log device busy at all times when under heavy load, so that log requests are batched while the disk is busy doing one write and these batched requests are sent immediately to the disk when the write finishes.

Hitz and associates [Hitz94] provide another point of reference that is perhaps closer to our approach. They included RAID-4 parity striping and NVRAM in their design. Their implementation uses shadow paging and makes use of a snapshot facility in combination with NVRAM logging of requests at the NFS level. Recovery consists of backing up to the most recent consistency snapshot and replaying the NFS log from that point. The only apparent drawback to this approach is that creating a snapshot involves modifying the entire block-map file and performing the housekeeping necessary to flush dirty data to the disk; the authors estimate that this task takes on the order of 1 second. Presumably, the recovery time when a system crashes at saturation could be at least as long as the time since the previous snapshot. We would not expect the NFS log to be replayed and processed faster after the reboot than when the original requests were seen, given that the server was in a saturated mode.

3 Netra NFS Design Features

The Netra NFS project started with the primary aim of producing an NFS server with very high performance, reliability and availability with the following design goals:

- High throughput with low latency, as measured by LADDIS [Whittle93]
- Tolerance of single disk failures without loss of file-system data
- Fast reboot after a power outage or system failure
- Fast disk and file-system initialization
- Minimal on-disk format changes
- Simplified browser-based administration

We give details of the features we implemented to satisfy these goals in the following sections: in section 3.1, we describe our NVRAM solution to provide low latency; in section 3.2, our use of Solstice Disk Suite to provide RAID-5 fault-tolerance; in section 3.3, our development of a cacheing technique to improve write performance; and in section 3.4, a modification to the UFS allocator to force locality of allocation for performance and to aid in fast recovery.

3.1 NVRAM Acceleration

It was clear from the beginning of the project that we required some form of NVRAM-based stable storage to satisfy the goals of high throughput and low response time [Moran90, Hitz94]. Originally, the entire memory of the machine was made nonvolatile by using an integrated uninterruptible power supply (UPS) that had sufficient backup power to shut down the system cleanly. Because we wanted to guarantee truly stable memory, we decided to use the Prestoserve [Presto93] accelerator to cache data. This approach provided us with a clean, debugged solution for fast stable storage. In recent versions, the UPS was been replaced by NVRAM boards.

3.2 Solstice Disk Suite

To satisfy the goal of tolerance for single disk failures, the Netra NFS server also includes RAID-5 and mirrored solutions based on the Solstice Disk Suite (SDS) product. In SDS, the update of a single arbitrary disk block within an industrial-strength RAID-5 device incurs 6 I/O operations: (1) read old data, (2) read old parity, (3) write new data to prewrite log, (4) write new parity to prewrite log, (5) write new data, and (6) write new parity. The prewrite log operations are necessary to preserve the atomicity of the new data and of the parity write operations, and to avoid the loss of data due to a system crash involving a disk failure. In the standard SDS implementation, the prewrite area is a reserved portion of the RAID-5 aggregate. To reduce the number of I/O operations needed, we decided to configure a second Prestoserve cache in addition to the one layered above the RAID metadevice, called Presto Upper. The second cache, Presto Lower, is configured below the RAID driver and completely covers the prewrite area of the RAID device. This cache eliminates two disk operations per random data-block write.

3.3 Bmap Cache

Our performance analysis of server systems under SPECsfs1.1 (SPECnfs_A93) LADDIS load revealed that write operations accounted for 35 to 50 percent of the time spent waiting for completion of NFS requests. We made traces while running LADDIS that showed that an average NFS write operation took 2.58 disk operations when NVRAM was not configured. We found that when a file grows beyond a certain size, the UFS file system has to write at least two disjoint pieces of metadata information: the inode and an indirect mapping block. Thus, when the write of the data is included, some NFS writes require three disk operations. We decided to look for a way to reduce the number of writes required. (The more recent SPECsfs2.0 version of LADDIS does not have this emphasis on writes to relatively large files; hence, improvements in this area are less critical now. However, in the context of SPECsfs1.1, our technique yields significant performance gains.)

Our solution, *bmap cache*, combines the disjoint metadata into one disk block, saving one I/O operation per write. The solution involved extending the file inode to cache that part of the indirect block that pointed to blocks at the end of the file. The inode and block-mapping information could then be updated atomically in one disk operation most of the time, rather than in two [Peacock96]. The simplest implementation would have been just to add the *bmap cache* fields to the inode itself, and to increase the size of the inode appropriately. However, since some utilities understand these structures and the layout of the file system, we decide not to change the size, layout, or location of inodes on disk. Instead, we stored the *bmap cache* in an inode extension that was separate from the main inode structure. The inode extension structure fits in a regular inode slot and looks like an unallocated inode. In the current implementation, file inodes are allocated from the even inodes, and an inode extension is allocated from the corresponding odd inode, in effect doubling the size of the inode.

With this allocation strategy, a *bmap cache* is stored on disk in the disk block that contains its inode. When a block is written at the end of a file to extend the file, the pointer to the new block is placed into the cache, rather than into the actual indirect block. Updating the metadata then involves writing the inode and its

extension to the disk in the same block; that requires only one I/O operation. When a block is allocated beyond the end of the cache, the cache is flushed to its actual indirect block, resulting in additional I/O, and the cache is set up to contain the mapping for the new block. In addition, although the indirect block is allocated when there are blocks that would be mapped by it, it is not initialized until the *bmap cache* is flushed.

3.4 Hot-Spot Allocator

In UFS, the file-system partition is divided into *cylinder groups*. Each cylinder group has a controlling structure residing in a single block that contains the resource bitmaps and is followed by an array of inodes. The size of each cylinder group is bounded by the size of the bitmaps that can be contained in a single block.

Part of the original purpose of cylinder groups was to allow locality of allocation for directory subtrees and to provide a mechanism to spread allocation across the disk. Each cylinder group has two allocation rotors that mark starting points to search circularly through the free-list bitmaps to allocate the next inode or block within the cylinder group. In addition, there is a single cylinder-group rotor that is used as the starting point to search for a lightly used cylinder group from which to allocate. Even with these rotors, it is possible for allocations to be distributed across the entire file system during any given short time period.

We have modified the rotor behavior to produce a *hot-spot allocator*, so-called because, at any given time, there is a single hot-spot cylinder group from which all allocations are done. Each individual allocation rotor starts at the beginning of its cylinder group when it is entered as the hot spot; when an allocation cannot be satisfied without wrapping either of the rotors back to the beginning of the cylinder group, the cylinder-group rotor is stepped to the next cylinder group, which becomes the new hot spot. We thus force temporal locality of reference in terms of inode and block allocation.

There are several good reasons to have these rotors move strictly sequentially through the file-system cylinder groups. Much recent work [Rosenblum92, Seltzer93, Seltzer95, Hitz94] has focused on high locality for file-system writes to relieve the perceived write bottleneck [Ousterhout89]. By forcing locality, the hot-spot allocator increases the chance for full stripe writes in RAID-5 disk arrays. Another advantage is that this locality of allocation makes it easier to keep track of the working set of the file system. We describe the relationship between the hot-spot allocator and consistency checking in section 5.2.

The hot-spot allocator also has a policy of slow reuse, since its allocation pointer increases relentlessly. This policy avoids certain race conditions that can cause loss of data when blocks are reused before pointers to them have been cleared. Although UFS is designed to avoid this situation, there have been observed instances (now fixed) where such races have occurred.

4 Reasons Not to Use Logging

We had already decided that NVRAM would be a key component of the system when we turned to solving the problem of fast recovery. We considered several log-based solutions before settling on fast consistency checking and found that existing log-based approaches did not give us sufficient performance, even with NVRAM enabled.

The first solution that we tried was the Solstice Disk Suite transaction logging for UFS. The LADDIS performance was so bad that it was difficult even to generate a valid LADDIS run, except at loads of less than 100 NFS ops. That is when we first observed how the synchronous nature of NFS requests fits poorly with a log-based approach. Hoping that perhaps the Veritas File System would perform faster when aided with NVRAM, we replaced UFS with it and analyzed its performance. Although it was not as bad as SDS logging, it still fell short of our performance goals. We considered implementing an NVRAM-based log for NFS requests, similar to [Hitz94], but decided that such a solution would not by itself solve the fast-recovery problem. We would still have to restore the underlying UFS file system to a consistent state after a reboot.

The key insight here is that *given enough CPU power, providing NFS service is a fundamentally disk-bound operation at saturation.* Each NFS request requires a certain number of I/O operations, so any technique that reduces or eliminates disk operations — such as the use of NVRAM and the *bmap cache* — will result in higher NFS throughput. Log-based solutions simply require more disk I/O, because every datum is written twice: once in the log and once in its actual place (except in log-based file systems, where the log *is* the actual place, as in the [Hitz94] approach). Adding NVRAM to a disk-based logging solution helps, but only by reducing effective disk response times. Such logs are typically written in a circular fashion — the worst case access pattern for a write cache. We have observed in our Veritas testing with NVRAM that dirty-write hit rates in the cache are only about 7%, whereas our chosen approach exhibits write hit rates between 60% and 70%.

Logging solutions are typically designed with disks, rather than NVRAM, as the target logging device. As such, there is an implicit requirement to batch logging information into a separate disk area so that the write cost to the disk can be amortized over a number of requests, and so that atomicity of transactions can be provided. When NVRAM is considered as an integral part of the logging design, the use of batching becomes unnecessary, because access time is independent of location. Thus, it seemed that an elegant approach to logging is simply to think of the Presto cache on top of the disk metadevice as the log for writes to the file system.

With no separate log area, it becomes helpful to include a small amount of additional information in certain file-system data structures. This extra state information allows us to undo partially written metadata transactions without having to do a full *fsck*. Since NFS operations are synchronous, the only partial transactions that should be found in the file system during recovery are those that were in progress. Correct operation of the server is ensured because the NFS client should retry the undone operation, since it has not received a completion acknowledgment. An additional benefit is that it also works in a degraded mode when the NVRAM is disabled due to a fault or to a low-battery indication. Data are simply written through to the actual disks with enough information to do fast recovery, since we do not insist on atomicity in the updating of the disjoint metadata involved in a transaction.

5 Fast Consistency Checking

The normal *fsck* program restores consistency of a UFS file system after an unclean shutdown by completely examining and fixing the following structures:

- Free block and inode bitmaps
- Directory consistency and inode link counts
- Allocation summary information

The resource-use bitmaps are reconstructed from the state of files at the time of reboot. A disk block is marked as allocated if it is referenced by an inode, which is in turn also referenced. Inodes that are referenced by directory entries or by shadow inode pointers are also marked as allocated, and their link counts are set to the number of references. (Shadow inode pointers are inode pointers within an inode that point to another inode; they are currently used to implement access control lists (ACLs)). All unreferenced blocks and inodes can then be marked free in the respective bitmaps. Inodes must also be reachable from the root directory of the file system to be marked as allocated. The summary counts of allocated and free resources are generated from the reconstructed resource bitmaps.

There are four changes to UFS that we implemented to facilitate fast recovery:

- Busy block and inode Bitmaps
- Inode linktags
- Cylinder-group flushing
- Last-inode counters

Figure 1 shows the layout of a single cylinder group showing the new elements we added and their locations, highlighted in italics.

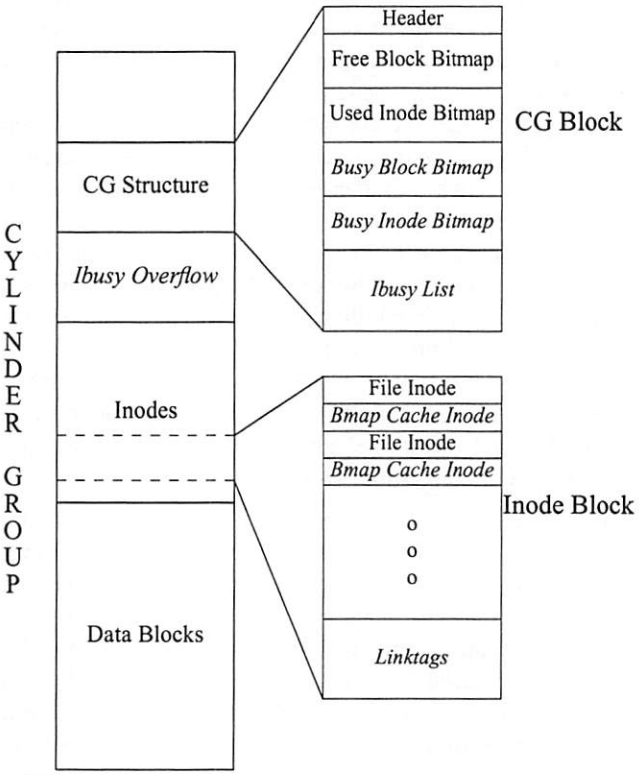


Figure 1. Cylinder Group Layout.

The busy block and inode bitmaps aid in recovering the free-block bitmaps, the inode linktags are used to do incremental repair of inode link counts, and cylinder-group flushing is used to keep low the quantity of dirty file-system metadata. The last-inode counter in each cylinder group is used to limit the number of inode slots that need to be initialized when the file system is created and looked at during a full *fsck*. In sections 5.1 to 5.4, we describe each of these components.

5.1 Busy Bitmaps

The first basic concept behind recovering the block-allocation bitmap without scanning the entire file system is to maintain a list of all blocks that are busy — that is, possibly in transition between the free and

allocated state — with reference to inodes that may or may not have been committed to stable storage. The second requirement is thus that a list be maintained of all inodes that are busy. The condition that must hold between these lists is that any block in the busy-block list either belongs to a file in the busy-inode list or is free. This condition allows the recovery algorithm to look at only those blocks owned by the busy inodes and to consider only those blocks that are in the busy-block list.

The busy-block list is implemented as a bitmap that is parallel to the free-block bitmap in each cylinder group. In a similar fashion, there is a busy inode bitmap parallel to the allocated-inode bitmap in each cylinder group. If each file could be contained in a single cylinder group, then the consistency of each cylinder group could be done independently with only these bitmaps. However, each cylinder group is limited to about 16 MBytes, so large files will span multiple cylinder groups. To handle this situation, and to enable each cylinder group to be checked individually, we gave each cylinder group an additional *ibusy* list containing inodes not in the cylinder group that have allocated or freed currently busy blocks in the cylinder group. The initial segment of the *ibusy* list is inside the cylinder-group block itself, with the adjacent block containing any overflow from that list. This is shown in Figure 1.

For this busy-block and inode approach to be workable, it is necessary to mark blocks and inodes busy before any actual state changes occur, and the marking must be done synchronously. Here, the hot-spot allocation strategy is useful. When a cylinder group is entered as the new hot spot, every free block in the cylinder group is marked busy, so this operation must be performed only once. Inodes are marked busy when they have blocks added or removed or during operations that modify their link counts. An inode number is added to the *ibusy* list of a cylinder group only when the first block from that cylinder group is allocated to it, so only one synchronous write of the cylinder-group information is required for all the allocations that the inode performs in the cylinder group.

5.2 Cylinder-Group Flushing

UNIX normally does periodic full flushing of the file system's modified metadata to get the file system into a "clean" state. The salient feature of the clean state is that the file-system metadata are consistent, so no *fsck* is required before a clean file system is mounted. One of our goals was to perform incremental cache flushing of the working set — particularly of the inode cache — because it is very difficult to get a file system into the clean state while under heavy load, and the clean state cannot be maintained for very long. Indeed, we found that LADDIS benchmark runs performed with the usual periodic cache flushing disabled performed significantly better.

Even though the clean state is infrequent and fleeting, we should take advantage of it. Reaching it essentially means that all metadata in the file system are consistent, and therefore all the busy bitmaps throughout the file system can be cleared. Rather than clear them by rewriting all the cylinder groups, we maintain in the file-system superblock a generation counter that is incremented whenever the clean state is reached. There is a corresponding counter in each cylinder group that is set to the value prevailing in the superblock whenever an operation affecting the busy bitmaps is performed. When the cylinder group and superblock counters are found not to match, the busy-bitmap information is known to be stale, and so is reset to a cleared state before the operation is performed. Similarly, on recovery with *fsck*, any cylinder group whose counter does not match the superblock's is considered to have stale busy information and thus does not need to be checked.

The recovery time for the busy bitmaps is proportional to the number of busy inodes, so a mechanism that tries to flush inodes and mark cylinder groups as clean is desirable. The busy bitmaps in combination with the busy-inode list on each cylinder group provide the ability to clean an individual cylinder group. We do so simply by flushing all the inodes in both the busy-inode bitmap and the *ibusy* list individually. The cylinder group is put into a *cleaning* state at the start, and, if an operation affecting the busy lists has not occurred after flushing of all the inodes, the cylinder group can be marked clean, which we do by setting its generation counter to be one less than the counter in the superblock.

The cylinder group flusher works like a two-handed paging daemon [Leffler89]. That is, the cylinder groups directly in front of the hot-spot cylinder group are flushed in a circular fashion. This approach provides the LRU properties that we would expect in such a clock-based pager. The flushing is conducted from a separate kernel thread that runs periodically and is set up to consume a fixed percentage of real time each time that it runs. With this approach, we can set the amount of time the flusher takes to a reasonably low value (10% of the flushing interval), and thus affect server performance only slightly.

5.3 Inode Linktags

A nonshadow inode's link count is simply the count of the number of directory entries that refer to the inode. It would be wonderful if the busy inode lists were sufficient to manage and recover inode link counts. However, they are not, because the link count on an inode before an operation is performed does not contain sufficient information to allow us to find the references to the inode without scanning the entire file system directory structure. To solve this problem, we incorporate some additional logical state, which we call a *linktag*, into the inode to enable undoing of a link-count increment or decrement when the corresponding directory update was not written to disk before a reboot. The *linktag* structure has the following form:

```
struct linktag {
        ino_t    dl_target;
        ino_t    dl_direct;
        off_t    dl_off;
        int      dl_cnt;
};
```

where the *dl_target* is the inode number of the inode whose link count has changed, *dl_direct* is the inode number of the directory that has had an entry added or removed pointing to *dl_target*, *dl_off* is the offset within the directory to a 512-byte directory block where the directory operation should occur, and *dl_cnt* is the expected number of entries pointing to *dl_target* within that 512-byte block (the 512-byte block is chosen because that is the size of the directory unit that UFS manipulates atomically). The count is necessary because a given directory block could contain multiple entries pointing to a given inode.

We use the *dl_cnt* and *dl_off* fields to make the creation and use of a *linktag* reasonably efficient. A search confined to a single directory block is much faster than a scan of a large directory for references to a given file. Setting *dl_cnt* to the expected count after the directory operation makes the recovery action simple. When *fast fsck* processes a *linktag* entry, it finds the actual count of references to *dl_target* at *dl_off* in *dl_direct*. *Fast fsck* then adjusts the link count of *dl_target* by adding the difference between the actual count and *dl_cnt*.

Since the *linktag* structure must be written to the file system synchronously with the updated link count in *dl_target*, we made use of the *bmap-cache* mechanism of reserving inode slots within the same inode block. In particular, the final one-eighth of each inode block is reserved to contain *linktag* entries. The *linktags* are kept as a list bound to the in-core *dl_target* inode and are placed into the reserved inode slots any time the *dl_target* inode is flushed to stable storage. The normal sequencing of operations requires that link-count increments are written before the creation of the directory entry and that decrements are written after the deletion of a directory entry. With the addition of the *linktag* state, the inode and its *linktag* must be written before the directory entry is changed. We thus require an additional write before a directory-entry deletion that was not done before, although this write is typically absorbed by the Presto Upper cache. The directory operations themselves are still done synchronously, and the *linktag* can be removed from the target inode after the directory block has been written. It is not necessary to rewrite the inode block with the *linktag* entry removed, because its *dl_cnt* reflects the true state of the directory. It will be cleared on the next flush of the inode to stable storage.

It is possible to have a single *linktag* entry involved in more than one operation. For example, suppose that there are two simultaneous link operations on a given file to the same block of the same directory. We handle this situation by associating a reference count with the *linktag* that prevents the *linktag* from being discarded until after all the operations using it have completed. The semantics of *dl_cnt* ensure that the link count will be recovered correctly, no matter in what state the multiple operations are when a crash occurs.

There is another subtle case that involves file renaming or linking: the link count is incremented before the *dl_off* value for the new directory entry is known. In this case, *dl_off* is set to an invalid offset of DL_NOSLOT that denotes that this *linktag* is a place holder. When a free slot in the directory has been obtained, the *linktag* is updated and is written to reflect the location of the new entry.

Directories also present a more complicated situation because each one has a ".." entry that points to its parent directory, and this link is reflected in the link count. Since there is only one ".." entry, we deal with this link also by setting an invalid offset value of DL_DOTDOT into the *dl_off* field.

Rename operations are complex — especially a move of a directory from its current owner to another directory already containing a directory with the same name. We handle such operations by breaking them down into individual *linktags* for each of the links that is created or removed. We did not implement an atomic rename operation because the *linktag* paradigm is not strong enough to support one. The biggest problem is that removing the old name of a renamed entity cannot be accomplished with only the *dl_cnt* and *dl_off* model of the *linktag*. It could be done if there were some

means of uniquely identifying each entry pointing to a given target. We could add a unique 1-byte tag to each directory entry pointing to a given target; we could then use the tag to identify which entry should be removed by *fsck* to complete the final stage of a rename.

5.4 Last Inode

One of the problems with enormous file systems is that a many empty inodes are typically configured when the file system is initialized. The standard UFS implementation requires that all these inodes be initialized during *mkfs*(1M), then examined during any full *fsck* operation. Dealing with such a large number of inodes dominates the time taken to create and *fsck* a file system. To reduce this time, a field giving the highest-numbered inode in use in each cylinder group, *lastino*, was added to each cylinder group. During a full *fsck* operation, this field allows the inodes examined to be limited to only those that might be used in each cylinder group. During file-system creation, this field allows us to bypass the initialization of all but the inode block containing the root inode, saving considerable time. The file system then initializes inode blocks as needed while the file system is in use. In particular, we found that making a new file system on a 40-GByte RAID-5 metadevice took 97 seconds with this feature enabled versus 1378 seconds with the standard UFS — a 14-fold reduction.

6 Performance Results

One of our goals was to increase the overall performance of NFS service provided by the Netra NFS server, as measured by the SPECsfs1.1 LADDIS benchmark [Whittle93]. LADDIS provides a client-independent characterization of a server's response time and throughput under a load consisting of direct remote calls to NFS service procedures on the server under test.

We performed a set of LADDIS benchmark runs to detail the effects on performance of enabling different aspects of our implementation. In each run, the server was loaded past the saturation point in steps. In addition, we crashed the server while it was under heavy load, and measured file-system recovery times under different configurations. The following elements were varied:

- RAID-5 (parity striped) versus RAID-0 (striped)
- Presto versus no Presto
- Bmap cache on versus off
- Fast fsck on versus off
- Veritas File System versus UFS

In addition, we performed a number of crash tests under various configurations and system loads to characterize the differences in recovery times.

6.1 Testing Configuration

The Netra NFS server used for benchmarking was an Ultra-2 with 2 300-MHz UltraSPARC(TM)-II processors with 1 GByte of DRAM and 32 MByte of NVRAM. The storage that we used for benchmarking was a SPARCStorage(TM) MultiPack configured with 12 4.2-GByte disks. The network controller used was the Sun Fast Ethernet (100Base-T). LADDIS load was generated by 2 Ultra-I machines, each configured with 64 MByte of DRAM and 167-MHz UltraSPARC processors.

The server was installed with the Netra NFS 1.2 release, which is essentially Solaris 2.5.1 with the UFS kernel module and utilities replaced by the Netra NFS-specific ones. The Sun Enterprise Volume Manager (Solstice Disk Suite) Version 2.4 provided RAID-0 and RAID-5 support. For the Veritas tests, the VxFS 3.2.2 File System and the VxLD 1.0.1 NFS Accelerator for Solaris were used. VxFS is normally configured with the transaction log contained at the beginning of the file system. An NFS load causes a lot of seeking between the log area and the actual file data, so Veritas developed VxLD to allow the log to reside on a device separate from the file system, thus avoiding the seek overhead. The VxFS plus VxLD configuration is thus similar to the approach described by Vahalia [Vahalia95].

For the RAID-0 and RAID-5 UFS tests, 11 drives were used. In the RAID-0 Veritas tests, 11 drives were set up as the file system, and a twelfth drive was designated as the VxLD log device. We set up all metadevices using an 8-KByte interleave size; in all cases, a single file system was created on the metadevice.

The Veritas setups would not strictly satisfy our design constraints for RAID-5 operation because the log disk is a single point of failure, and it would have to be mirrored to avoid loss of data should a log disk fail. This mirroring would decrease performance, but these results provide an approximate comparison. Also, we gave the Veritas RAID-5 setup the benefit of an extra disk drive in the tests.

6.2 LADDIS Results

The two factors that most influence performance are the choice of RAID-0 over RAID-5 and the presence or absence of Presto NVRAM cacheing. Figure 2 shows a set of LADDIS response-versus-throughput curves for these four combinations, with both *bmap*

cache and *fast fsck* enabled. In addition, there is a plot with Presto Lower disabled, (Presto Upper only), to show that component's effect on the RAID-5 results. No benefit accrues from configuring Presto Lower in a RAID-0 configuration. Presto increased the LADDIS *figure-of-merit number* (maximum throughput, and response time at that throughput) for RAID-5 from 303 NFS ops. at 61.7 msec to 1397 NFS ops at 13.8 msec and for RAID-0 from 859 NFS ops. at 44.4 msec to 3046 NFS ops at 20.0 msec.

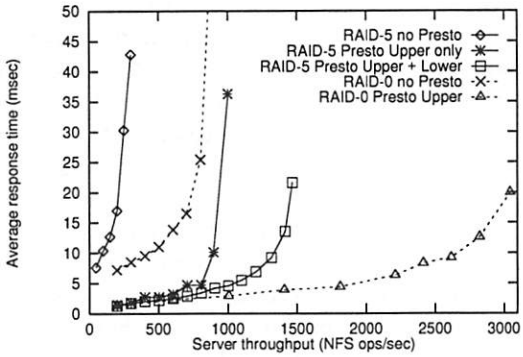


Figure 2. Comparison of RAID and Presto combinations.

Figures 3 through 6 show more detailed sets of LADDIS response-versus-throughput curves for benchmark runs done with various combinations of features enabled. In general, the *fast fsck* feature caused an increase in disk activity and a decrease in throughput, whereas the *bmap cache* decreased the disk activity and increased the throughput. Enabling Presto reduced the size of these differences.

The Veritas results illustrate why we chose not to use an existing disk-based logging solution with Presto added. In Figure 3, from the curves obtained with Presto disabled, we see that the VxFS/VxLD combination tracks the Solaris baseline plot closely up to 400 NFS ops., although it is not as good as the *bmap-cache* plot up to 500 NFS ops.; but then is better above those points, and actually has the highest throughput of 587 NFS ops. at 58.2 msec. The plot for *fast fsck* alone shows relatively poor performance, reaching only 250 NFS ops. at 43.4 msec, and saturating at 298 NFS ops at 94.2 msec.

Figure 4 shows the normal mode of operation with RAID-5 and Presto enabled; here, there is a complete reversal. The Veritas plot shows the lowest performance, reaching only 747 NFS ops. at 49.3 msec. This is still better better than the no-Presto case, but it is far below the case of *bmap cache* plus *fast fsck* (1466 NFS ops. at 21.6 msec). The benefit of the *bmap cache* can be seen clearly in Figure 4. The maximum throughput increases by 23% when the *bmap cache* is added to the *fast fsck* configuration.

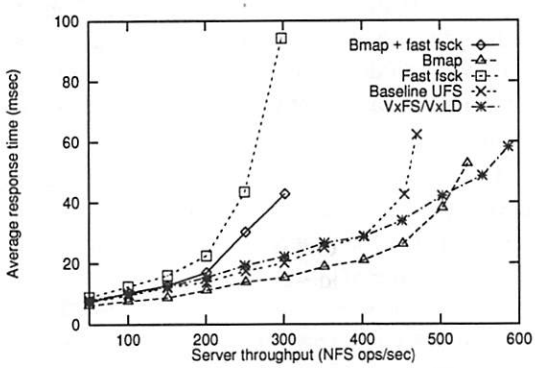


Figure 3. RAID-5 LADDIS performance without Presto.

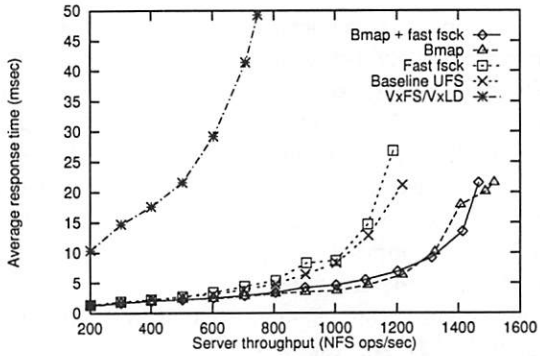


Figure 4. RAID-5 LADDIS performance with Presto.

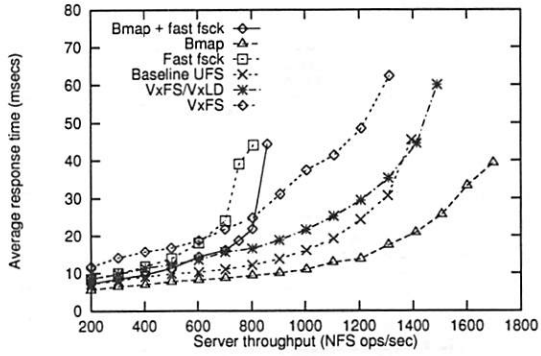


Figure 5. RAID-0 LADDIS performance without Presto.

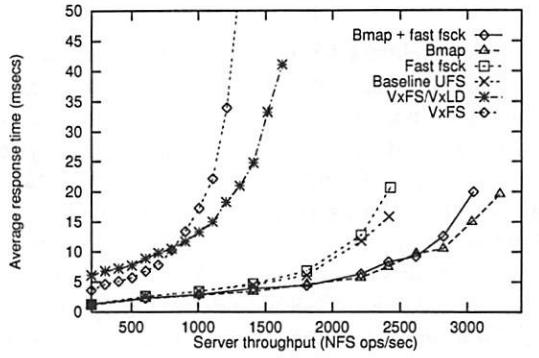


Figure 6. RAID-0 LADDIS performance with Presto.

The RAID-0 results, shown in Figures 5 and 6, are similar, although of course the throughput scales are larger. We included plots of VxFS with and without VxLD to show the effect of moving the VxFS log to a separate disk.

Most of the runs in the results shown in the graphs were repeated multiple times; the results were stable. To get an estimate of the sampling error, we did 10 runs of an 11-disk RAID-0 configuration at 3000 NFS ops. The average response time at that load was 10.92 msec, with a standard deviation over the 10 runs of 1.05 msec.

6.3 Crash-Test Results

To obtain a characterization of the recovery performance, we deliberately and repeatedly crashed the NFS server under test 5 minutes into the second of two 10-minute LADDIS runs, at varying client loads. We used the first run to measure throughput and response time, so that we could determine the point at which saturation occurred. We collected data in the 11-disk RAID-0 configuration using standard UFS as a baseline, *bmap cache* alone, *bmap cache* plus *fast fsck* with some variations on its tuning parameters, and VxFS without VxLD. All tests were run with Presto enabled. The results are shown in Figure 7, where we note many interesting points.

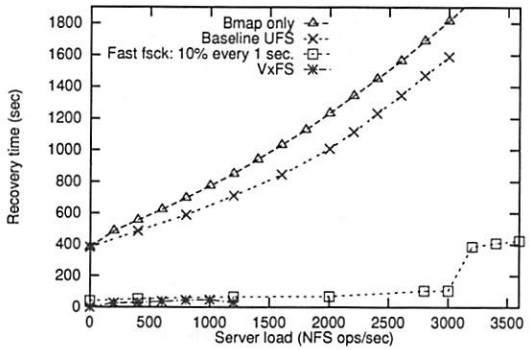


Figure 7. Recovery time versus server load .

The baseline UFS plot is almost a straight line, increasing as a function of load. This almost linear rise occurs because the *fsck* time is proportional to the number of files and allocated blocks in the file system, which in turn is directly proportional to the LADDIS load. The constant portion — where the load is 0 and the file system is empty — reflects that *fsck* has to read all the cylinder groups and inodes at least once. This takes 382 seconds in our 44-GByte file system. The *bmap* plot shows higher recover times than baseline UFS and this is probably because *fsck* writes *bmap-cache* indirect-block information out to the correct place in the indirect block. We made the VxFS plot without using VxLD because we encountered difficulty with VxLD. The VxLD enhancement moves the log off the file system proper onto a separate device, but on reboot copies the log data back into the file system. The VxFS recovery then proceeds as though the data had been logged into the file system originally. This copying typically took between 45 and 50 seconds, so it should be added to the VxFS recovery times if VxLD is used. The VxFS recovery times are otherwise good, with no overhead on an empty file system.

The *fast fsck* result is between the baseline UFS and VxFS numbers, as might be expected. There is a constant overhead portion of about 42 seconds that occurs because *fast fsck* has to read all the cylinder groups at least once. The main reason that it does so is to maintain consistency of allocation counts in the file-system superblock and in the cylinder-group summary area just after the superblock. It would be possible to maintain a busy-cylinder-group bitmap within the superblock and only visit those cylinder groups marked busy. Doing that would reduce the empty or clean file-system overhead to 0, and probably would lower the *fast fsck* recovery time below that of VxFS, although it would complicate maintaining consistency of the summary area.

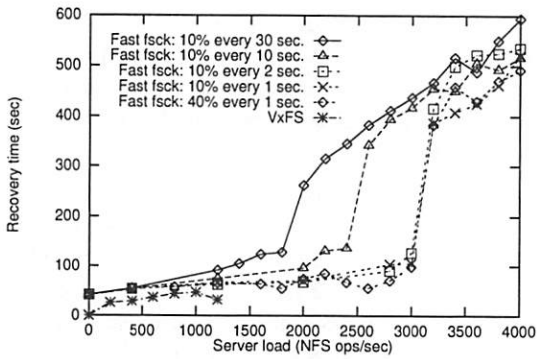


Figure 8. Flushing-rate effect on *fast fsck* times.

The rate and duty cycle of cylinder-group flushing have an effect on the shape of the recovery-time curves as a function of the tuning parameters, as shown in Figure 8. There are two parameters: the percent of real time that the flushing thread is active, and the interval between activations. The default values were set such that the flusher runs every 30 seconds for 10% of the interval, or for 3.0 sec. We see that, as the load increases, the recovery time rises, because the higher load on the server allows more inodes and cylinder groups to become busy between flushes. Decreasing the interval between flushes yields a significant reduction in recovery time at intermediate loads.

Running the flusher every 1.0 sec allows the recovery time to be flat up to the saturation point at

3000 NFS ops., indicating that it is doing a good job of keeping the file system mostly clean. Once the load is beyond saturation, however, the flusher seems unable to clean the file system sufficiently fast, probably because it does not get enough disk bandwidth to clean effectively. The flusher does only one disk operation at a time, so that it slows dramatically when the disk queues become large. The recovery time curve for using a 40% duty cycle every second was nearly identical to the 10% every second curve, so increasing the flusher's active time did not give any benefit.

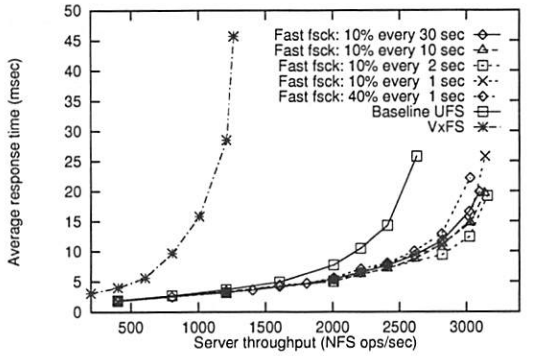


Figure 9. Flushing-rate effect on throughput and response time.

Figure 9 shows the corresponding LADDIS throughput curves for the tuning values shown in Figure 8, with the VxFS response curve shown for comparison. Using VxFS, for about a 60% reduction in maximum throughput, we can achieve a 25% to 50% reduction in recovery time between *fast fsck* and VxFS. Notice that the *fast fsck* response curves are virtually identical across the range of tuning parameters, although the 40% duty-cycle plot is above the others.

6.4 Comparison with Other Approaches

The LADDIS numbers reported by Vahalia [Vahalia95] reach a peak of 597 NFS ops. at around 29 msec of latency on a 60-MHz Pentium platform with 11 disks. In comparison, their nonbatching logging implementation peaks at around 400 NFS ops. Vahalia's approach is clearly viable, and might exhibit better performance using an NVRAM-based logging device, although a Presto-based solution would likely behave similarly to VxFS/VxLD. The authors measured reported recovery times by crashing the system at peak LADDIS load of around 600 NFS ops., at which time their file system had about 3 GByte of data over 10 (logging case) or 11 (no-logging case) disks. The recovery times were 450 sec for the standard *fsck* case, and between 3 and 14 sec for log-based recovery. The authors mention that the standard *fsck* checking was done serially, implying that their disks contained separate file systems.

Bearing in mind that the architectures of our system and Vahalia's are very different in terms of CPU, bus, memory, and controller configuration, we still attempted to obtain an approximate comparision with their results. We created a single file system of about 5 GByte on a RAID-0 device with 11 disk drives. The runs used for comparison had *bmap cache* and *fast fsck* enabled, and timings were done both with and without Presto enabled. We measured an average response time at 600 NFS ops. of 12.4 msec; saturation occurred at about 874 NFS ops. and 38.1 msec. Our higher throughput and lower latency is probably due to the combined benefits of using the *bmap cache* and more powerful processors. With Presto enabled and a necessarily larger file system, the response time at 600 NFS ops. was 2.2 msec, and saturation occurred at about 3000 NFS ops at 16 msec.

On a file system populated by a 600 NFS ops. LADDIS run and unmounted cleanly, *fast fsck* ran in 4 sec, which would be the best case on a lightly loaded system after a crash. Table 1 compares recovery times under several scenarios when the server was crashed at 600 NFS ops.

File System	Recovery Time (sec)	Standard Deviation
Vahalia	3—14	—
fast fsck + Presto	15.6	4.4
VxFS + Presto	34.3	3.1
VxFS	51.9	6.5
fast fsck	54.0	10.8
UFS	219.0	8.9

Table 1. Comparison of Recovery Times

As we might expect, the *fast fsck* recovery times without Presto are longer than are those achieved with Vahalia's logging approach, because the information maintained to do the recovery is less precise than a transaction log. On the other hand, our throughput and latency numbers appear to be better even without Prestoserve enabled and the recovery times are competitive with Presto on. It may be surprising that *fast fsck* with Presto takes less time to do recovery than does VxFS on a small file system, but consider that the relative overhead for *fast fsck* to read all the cylinder groups is lower with fewer cylinder groups.

To get a comparison with Hitz's results, [Hitz94], we used benchmark results from the SPEC web page at *http://www.spec.org/osg/sfs93/results/results.html* for the Network Appliance Corporation F520 and F540. These systems use a 275 MHz Alpha processor with 14 disk drives in a RAID-4 configuration. The maximum

throughputs are 2361 NFS ops at 8.3 msec for the F520 and 2230 NFS ops. at 7.7 msec for the F540. If we multiply our RAID-5 throughput number of 1397 by 13/10 to scale for the larger number of data drives, we get 1905 NFS ops. as a comparable extrapolation at 13.8 msec. It is likely that the difference between these results is due to lower RAID-4 parity overhead afforded by the log-structured WAFL file system, which is designed to do writes in full stripe widths.

7 Conclusions and Additional Work

The benchmarks for crash-recovery times as well as those for the server performance under LADDIS loads indicate that UFS with the fast consistency checker is a highly competitive local file system for use in an NFS server. Since our approach was designed with NVRAM as a key component, we were able to achieve performance that was significantly better than that obtained with logging approaches with NVRAM added as an afterthought, and we maintained acceptable crash-recovery times. The *fast fsck* consistency checking was 4 to 30 times faster than the original UFS *fsck*, depending on the configuration and on the load when the system went down.

An additional avenue of exploration is to determine the effectiveness of the hot-spot allocator when a parity-block cache is added to the RAID-5 implementation. We expect this cache to improve the single-writer write throughput of the system, and to increase the maximum LADDIS throughput. We are now characterizing the system's behavior with SPECsfs2.0.

8 Acknowledgments

A few people who worked with us on developing the ideas and implementation described here deserve mention: John Corbin was involved early in the architectural discussions surrounding how we should use the NVRAM effectively, and Rob Gittens worked on the changes to SDS that supported Presto Lower. Also, the reviewers comments were very helpful in improving the paper, and Pei Cao's shepherding of the paper in particular was most appreciated. Finally, we would like to thank Lyn Dupre′ for her expert and efficient help in editing the paper.

9 References

[Chutani92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, R. Sidebotham, "The Episode File System," *Proceedings of the USENIX Winter 1992 Conference*, pages 43-59, January 1992.

[Ganger94] G. Ganger, Y. Patt, "Metadata Update Performance in File Systems," *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49-60, November 1994.

[Hagmann87] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *ACM Symposium on Operating Systems Principles*, pages 155-162, November 1987.

[Hitz94] D. Hitz, J. Lau, M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the USENIX Winter 1994 Conference*, pages 235-245, January 1994.

[Juszczak94] C. Juszczak, "Improving the Write Performance of an NFS Server," *Proceedings of the USENIX 1994 Winter Conference*, pages 247-259, January 1994.

[Leffler89] S.J. Leffler, M. McKusick, M.J. Karels, J.S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System* Reading, MA: Addison-Wesley, 1989.

[McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, **2**(3):181-197, August 1984.

[McKusick94] M. McKusick, T.J. Kowalski, "Fsck — The UNIX File System Check Program," *4.4 BSD System Manager's Manual* Sebastopol, CA: O'Reilly & Associates, 1994.

[Moran90] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, B. Lyon, "Breaking Through the NFS Performance Barrier," *Proceedings of the 1990 Spring European UNIX Users Group*, April, 1990.

[NFS94] B. Pawlowski, C. Juszczak, P Staubach, C. Smith, D. Level, D. Hitz, "NFS Version 3: Design and Implementation," *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 137-151, June 1994.

[Ousterhout89] J.K. Ousterhout, F. Douglis, "Beating the I/O Bottleneck: A Case for Log-Structured File Systems," *Operating Systems Review*, **23**(1):11-27, January 1989.

[Ousterhout90] J.K. Ousterhout, "Why aren't Operating Systems Getting Faster as Fast as Hardware?" *Proceedings of the USENIX Summer 1990 Conference*, June 1990.

[Patterson88] D. Patterson, G. Gibson, R. Katz, "A Case for Redundant Arrays of Inexpensive Disks

(RAID)," *ACM SIGMOD Conference*, June 1988.

[Peacock96] J.K. Peacock, "Method and Apparatus for Caching File Control Information", U. S. Patent Application Serial No. 08/673,958.

[Presto93] Digital Equipment Corporation (DEC), *Guide to Prestoserve*, DEC OSF/1 Prestoserve Product Documentation, (Order number AA-PQTOA-TE), March 1993.

[Rosenblum92] M. Rosenblum, J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1-15, Association for Computing Machinery SIGOPS, October 1991.

[Sandberg85] R.D. Sandberg, S. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," *Proceedings of the USENIX Summer 1985 Conference*, June 1985.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the Winter 1993 USENIX Conference*, pages 307-326, January 1993.

[Seltzer95] M. Seltzer, K. Smith, H. Balarkishnan, J. Chang, S. McMains, V. Padmanabhan, "File System Logging Versus Clustering: A Performance Comparison," *Proceedings of the USENIX 1995 Conference*, pages 249-264, January 1995.

[Vahalia95] U. Vahalia, C.G. Gray, D. Ting, "Metadata Logging in an NFS Server," *Proceedings of the USENIX 1995 Conference*, pages 265-276, January 1995.

[Wittle93] M. Wittle, B. Keith, "LADDIS: The Next Generation in NFS File Server Benchmarking," *Proceedings of the USENIX Summer 1993 Conference*, pages 111-128, June 1993.

10 Author Information

J. Kent Peacock, Ph.D., has worked as an architect and implementor on the Netra NFS project since 1995. Between 1990 and 1997, he worked as a consultant for SMCC, SunLabs, SunSoft, Veritas, Hewlett-Packard, and Intel, generally doing multiprocessor implementations and file-system improvements. While at Intel, he worked on the multiprocessor version of SVR4 that was released by USL as SVR4/MP. Prior to becoming a consultant, Dr. Peacock rolled the dice at two startup companies: Dialogic Systems and Counterpoint Computers. He earned his B.Sc. in electrical engineering from the University of Manitoba, Canada, and then Masters and Ph.D. degrees in computer science from the University of Waterloo, Canada, where he met and fell in love with UNIX in 1978. When he is not trying to make computers do things faster, he spends time trying to lower his golf handicap of 11 or playing in the Long Day Flute Quartet and the HP Symphony.

Ashvin Kamaraju, M.A., M.S., is the engineering manager for Netra file-server and Proxy-cache server products at Sun. He has been at Sun for the past 7 years, where he has worked on the stable memory allocator and almost all the subsystems of the Netra NFS server. Prior to joining the file-server project, he worked in the desktop-systems group at Sun, doing operating-system porting, memory-management software and virtual-memory drivers for graphics acclerators. He earned his M.A. in mathematics and computer science and his M.S. in chemical engineering from the University of Cincinnati.

Sanjay Agrawal, M.S., is an engineer in the Netra NFS server group at Sun, where he has been working on NFS server performance and benchmarks. He holds an M.S. in computer science and a B.S. in electrical engineering from Indian Institute of Technology at Kanpur, India.

All the authors can be reached by U.S. Mail at Sun Microsystems, 901 San Antonio Rd., MS UMPK03-119, Palo Alto, California 94303, or by electronic mail at {kent,akk,sanj}@eng.sun.com.

General Purpose Operating System Support for Multiple Page Sizes

Narayanan Ganapathy
Curt Schimmel

Silicon Graphics Computer Systems, Inc.
Mountain View, CA
{nar, curt}@engr.sgi.com

Abstract

Many commercial microprocessor architectures support translation lookaside buffer (TLB) entries with multiple page sizes. This support can be used to substantially reduce the overhead introduced by TLB misses incurred when the processor runs an application with a large working set. Applications are currently not able to take advantage of this hardware feature because most commercial operating systems support only one page size. In this paper we present a design that provides general purpose operating system support that allows applications to use multiple page sizes. The paper describes why providing this support is not as simple as it first seems. If not designed carefully, adding this support will require significant modifications and add a lot of overhead to the operating system. The paper shows how our approach simplifies the design without sacrificing functionality and performance. In addition, applications need not be modified to make use of this feature. The design has been implemented on IRIX 6.4 operating system running on the SGI Origin platform. We include some performance results at the end to show how our design allows applications to take advantage of large pages to gain performance improvements.

1 Introduction

Most modern microprocessor architectures support demand paged virtual memory management. In such architectures, a process address space is mapped to physical memory in terms of fixed size *pages* and an address translation is performed by the processor to convert the virtual memory address to a physical memory address. The translation is done by traversing a *page table*. The processor performs the virtual to physical address translation on every memory access and hence, minimizing the translation time is of great importance. This is especially true for processors with physically tagged caches that reside on the chip because the address translation has to be completed before data can be retrieved from the cache. (A thorough treatment of caches on uniprocessor and multiprocessor systems can be found in [Schim94]). To achieve this goal, most microprocessors store their recently accessed translations in a buffer on the chip. This buffer is called the *translation lookaside buffer* (TLB). If a translation is not found in the TLB, the processor takes a *TLB miss*. Some processors like the i860 have built-in hardware to walk the page tables to find the missing translation, while others like the MIPS R10000 generate an exception and a TLB miss handler provided by the operating system loads the TLB entries.

The number of entries in a TLB multiplied by the page size is defined as ***TLB reach***. The TLB reach is critical to the performance of an application. If the TLB reach is not enough to cover the *working set* [Denn70] of a process, the process may spend a significant portion of its time satisfying TLB misses. The working set size varies from application to application. For example, a well tuned scientific application can have a small working set of a few kilobytes, while a large database application can have a huge working set running into several gigabytes. While the working set size can be reduced to a certain extent by tuning the application, this is not always possible, practical or portable. Hence, if a system has to perform well running a variety of applications, it has to have good TLB reach. Modern processor caches are getting large (> 4MB) and few microprocessors have a TLB reach larger than the secondary cache when using conventional page sizes. If the TLB reach is smaller than the cache, the processor will get TLB misses while accessing data from the cache, which increases the latency to memory.

One way to increase the TLB reach is to increase the page size. Many microprocessors like the MIPS R10000 [Mips94], Ultrasparc II [Sparc97] and PA8000 [PA-RISC] now use this approach. By supporting selectable page sizes per TLB entry, the TLB reach can be increased based on the application's working set. For example, the MIPS R10000 processor TLB supports 4K, 16K, 64K, 256K, 1M, 4M and 16M page sizes per

TLB entry, which means the TLB reach ranges from 512K if all entries use 4K pages to 2G if all entries use 16M pages. As a different page size can be chosen for each TLB entry, a single process can selectively map pages of different sizes into its address space. The operating system can choose pages of larger sizes for an application based on its working set.

While many processors support multiple page sizes, few operating systems make full use of this feature because providing such a feature has its challenges. Current operating systems usually use a fixed page size. The virtual memory subsystem is highly dependent on the fact that the page size is constant. For convenience, let us define this page size as the *base page size* and define *base pages* to be pages of base page size. Let us also define pages larger than the base page size to be *large pages*. The functions of the VM subsystem like allocating virtual addresses, memory read/write access protections, file I/O, the physical memory manager that allocates memory, the paging process, the page tables, etc., assume that there is only one page size which is the base page size. Additionally, many VM and file system policies like the page replacement policy, read ahead policy, etc., also assume a fixed page size. The file system manages all its in-core data in pages of the base page size. As we shall see later, modifying the file system and the I/O subsystem to support multiple page sizes is a formidable task. The design is further complicated by the fact that many processes typically share physical memory when they map files using the UNIX `mmap()` system call or when they use shared libraries. This means that two processes might share the same physical memory but may need to map them with different page sizes. To complicate matters further, the processor also adds several restrictions to the use of this feature.

Due to the invasive nature of the operating system modifications needed to support them, large pages have been put to limited use. They have only been used in special applications to map frame buffers, database shared memory segments, and so on.

General purpose support for multiple page sizes would go a long way in improving the performance of a substantial number of applications that have large working sets. In this paper we present a design that provides general purpose operating system support for multiple page sizes. An important characteristic of the design is that it adds no overhead to common operations and there is no performance penalty for applications when not using large pages. Another important feature of this design is that the knowledge of various page sizes is restricted to a small part of the VM subsystem. The file system, the I/O system and other subsystems are not aware of multiple page sizes. More importantly, large pages are transparent to applications and they need not be rewritten to take advantage of large pages.

The design has been implemented in SGI IRIX 6.4 running on the SGI Origin 2000 platform which uses the MIPS R10000 microprocessor. The following sections concentrate on the details of MIPS R10000 and IRIX 6.4, but the techniques presented here can be applied to most commercial processors and operating systems. Unless specified explicitly, the base page size is assumed to be 4K bytes.

Section 2 describes some related work in the area. Section 3 explains the microprocessor support for multiple page sizes, specifically describing the MIPS R10000 TLB. Section 4 describes the current IRIX VM architecture. Section 5 describes our design goals. Section 6 describes in detail our design and section 7 shows performance results for some sample benchmarks.

2 Related Work

The paper by Khalidi et. al [Khal93] studies the issues involved with providing multiple page size support in commercial operating systems. Although the study finds that such a support is non-trivial to implement, it does not propose any methods or algorithms to help provide this feature. The paper by Talluri et. al [Tall94] describes a new hardware TLB architecture that considerably reduces the operating system support. It proposes a new feature called *sub-blocking* and goes on to show how a sub-blocking TLB simplifies the operating system work involved while providing performance benefits similar to those provided by large pages. Sub-blocking is not yet provided by popular commercial microprocessors. The paper by Romer et. al [Romer95] describes different ***on-line page size promotion*** policies and how effective they are in reducing the TLB miss cost. For the policies to be effective, the operating system should maintain TLB miss data per large page. The cost of collecting the data in our implementation is significantly higher than what is mentioned in the paper. In addition, one of our important goals is to not penalize applications that do not use large pages until we fully understand the practical benefits of using large pages on commercial applications. Collecting the TLB miss data to do page promotion will affect the performance penalty of all processes and does not meet our goal. We are planning to do on-line page promotion in one of our future releases.

3 Microprocessor Support

The MIPS R10000 processor [Mips94] TLB supports 4K, 16K, 64K, 256K, 1M, 4M and 16M page sizes. Figure 1 shows a sample TLB entry. The MIPS R10000 TLB has 64 entries. Each TLB entry has two *sub-entries*. The subentries are related to one another in that they map adjacent virtual pages. Both pages must be of the same size. Thus, each TLB entry maps a virtual address range that spans twice the chosen page size.

The TLB entry also carries a *TLB PID* that identifies the process to which the TLB entry belongs. On a TLB miss, the processor generates an exception. The exception invokes the software TLB miss handler which inserts the appropriate TLB entry. The software TLB miss handler is part of the operating system.

Like most microprocessors, the R10000 TLB restricts the alignment of the virtual addresses and physical addresses that can be mapped by a large page to the large page size boundary. For example, a 64KB page can be mapped only to a virtual address aligned to a 64KB boundary. The physical address of the large page must also be aligned to a 64KB boundary. The virtual address that maps the first of the two sub-entries in the TLB, called the even sub-entry, must be aligned to a *2*pagesize* boundary. The virtual address range covered by the large page must have the same protections and cache attributes if the entire range is to be mapped by one TLB entry.

4 Basic VM Structure

Almost all commercial operating systems provide virtual memory support [Cox94]. Most use pages of only one page size and all major subsystems manage their memory in pages of this size. These include the memory management subsystem that maps virtual to physical memory for a given process, the file system, the I/O subsystem and the machine dependent portion of the operating system kernel. The IRIX VM subsystem has two main components:

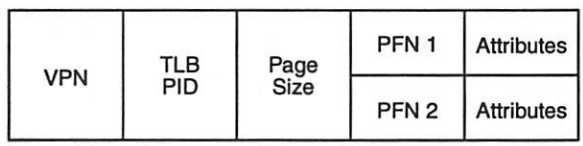


Figure 1: MIPS R10000 TLB Entry Format.

A *physical memory manager* that manages memory in terms of page frames of a fixed size. Each physical page frame in the system is represented by a kernel data structure, called the page frame data structure or the *pfdat*. The physical memory manager provides methods for other systems to allocate and free pages.

A *virtual memory manager* manages the process's address space. An address space consists of several segments called *regions*. Each region maps a virtual address range within the address space. The regions are of different types. For example, the *text* region maps a process's text and the *stack* region maps the process stack. Each region is associated with a particular memory object. For example the text region's memory object is the file containing the process text. The data for the text region is read from the file into physical memory pages and they are mapped into the process address space. Each memory object has a cache of its data in physical memory. The cache is a list of pfdats that represent the physical pages containing that object's data. For this reason the cache is called the *page cache*. Thus the physical pages containing a file's data are kept in the file's page cache. A memory object can be shared across multiple processes. The file system also performs its read and write requests in terms of pages. The device drivers do their I/O in terms of pages.

The virtual address space is mapped to physical page frames via page tables. There are numerous choices for the page table formats and several have been discussed in the literature [Tall95]. The page table format for IRIX is a simple forward mapped page table with three levels. Each page table entry (PTE) maps a virtual page to a physical page. The PTE contains the page frame number (PFN) of the physical page that maps the virtual page number. It also contains bits that specify attributes for the page including the caching policy, access protections and of course, the page size.

The virtual memory manager supports *demand paging*. The procedure of allocating the page and initializing it with data from a file or the swap device is called *page faulting*.

5 Design Goals

The overall goal is to provide general purpose multiple page size support. This includes dynamic allocation of large pages, faulting them in to a process address space, paging them out, and upgrading and downgrading page sizes. Our emphasis is to make the design simple and practical without compromising functionality and performance.

Applications should be able to use large pages without any significant modifications to their code. System calls which depend on the page size should behave the same way when using large pages as they would when they use base pages.

Application performance should degrade gracefully under heavy memory load. They should at least perform as well as they would when not using large pages.

As indicated in the previous sections, blindly making the extensions to all the OS subsystems that use the knowledge of page size is an enormous task. It is not only difficult to implement but also introduces performance penalties in many key parts of the operating system and slows down applications that do not use different page sizes. It is important to understand that large pages only benefit applications with large working sets and poor locality of reference. Not all applications will need large pages. One of the fundamental goals of the design is to not penalize the performance of applications that do not use large pages. Another goal is to restrict the extensions needed to as small a set of OS subsystems as possible.

6 Our Design

This section describes our approach in detail. We decided to retain the original format of pfdat and PTE data structures. As we see later, these two choices are crucial to the success of our design. The remaining sections describe the TLB miss handler, the large page allocator, the page fault handler, the page replacement algorithm, the algorithms to upgrade and downgrade page sizes and the policies we use with large pages.

6.1. Pfdat Structure

One of the first design decisions to be made is how to handle the pfdat structures for large pages. A simple-minded approach might be to make a pfdat represent a large page frame by adding a page size field to the pfdat. But as we will soon see, this approach is fraught with problems.

The pfdat structure is a very basic data structure and is used widely by the VM, file system and the I/O subsystem. It is the representation of a physical page frame and tracks the state changes that happen to a page. Changes to the pfdat structure will mean changes to all the subsystems that use the data structure. Traditionally pfdats are an array of small structures. Hence the conversion functions from physical page frame numbers to pfdats and vice versa are simple and fast. If pfdats represent pages of different sizes the conversion functions become more complex.

There is an impact on the page cache data structure as well. The data structure is usually a hash table and the pfdats are hashed into the table by a hash function that takes the memory object address and the offset into the memory object. The hash function works well if the pfdats are of fixed size since the pfdats will be distributed evenly across the hash buckets. If pfdats represent pages of different sizes, the simple hash functions do not work very well. A more complicated hash function will slow down the search algorithms that are crucial to system performance.

Another problem is that processes sharing a set of physical pages would have to be able to map these pages with different page sizes. For example, suppose one pfdat represents a large page and it is shared by a set of processes. If one of the processes decides to unmap part of its virtual address space mapped by the large page, the large page would have to be split into smaller pages. In addition, the page table entries in all the processes that map the large page would have to be downgraded as well. If the large page belongs to a heavily shared memory object like a library page, the downgrading operation would incur a significant performance penalty.

From the preceding discussion, it is clear that the disadvantages of having pfdats represent page frames of different sizes are too great. Our design therefore chooses to retain the original structure for pfdats i.e., pfdats represent pages of a fixed size (the base page size). Large pages are treated as a set of base pages. Thus if the base page size is 4K and we use a 64K page to map a segment of a file, there will be sixteen pfdats in the page cache to map that segment.

This essentially means that most subsystems that interact with the pfdats do not have to know about large pages. It enables us to avoid rewriting many parts of the operating system kernel that deal with lists of pfdats. These include the file system, the buffer cache, the I/O subsystem and the various device drivers, considerably simplifying the complexity of providing multiple page size support. For example, if the contents of a 64K large page has to be written to the disk, the device driver is passed a buffer that contains 16 4K pfdats corresponding to the large page. The device driver does not know it's a large page and handles the buffer just like it would handle any other buffer that has a list of pages. Of

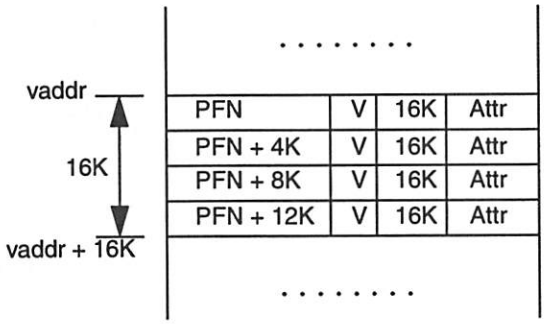


Figure 2: Multiple Page Table Entry Format.

course for the large page case, the device driver can be smart enough to recognize that the pages are contiguous and can make a single DMA request.

This decision also means that the lowest page size we can ever use in the system will be the base page size. In practice, this is not such a severe restriction as the base page size is usually the smallest page size supported by the processor. In the next section, we see how a set of ptdats are treated as a large page even though the pfdats themselves do not carry any page size information.

6.2. Page Table Entry Format for Large Pages

The next important decision we need to make is regarding the format of the page table. The page table entries (PTEs) need to contain the page size information so that the TLB miss handler can load the right page size into the TLB entry.

We decided to retain the existing page table layout. At the page table level we have one PTE for each base page of the large page. Figure 2 shows how the PTEs would look for a 16K large page with a 4K base page size.

In this format, we use all the PTEs that span the large page range. The PTEs that map large pages look similar to those that map base pages. For the large page PTEs, the page frame numbers will be contiguous as they all belong to the large page. In addition, all PTEs have a field that contains the page size. The TLB miss handler needs to look at only one PTE to get all the information needed to drop in the TLB entry. As the information is present in each PTE, the handler traverses the page table just like it does for the base page size case. It finds the appropriate PTE and drops the contents into the TLB. As such, the large page TLB miss handler performance is comparable to the performance of the base page size TLB miss handler (described in section 6.3). This approach also helps while upgrading or downgrading the page size. Only the page size field in the PTE needs to be changed.

One important benefit to this format is that the subsystems that deal with PTEs need not be modified as the PTE format remains the same. There is an additional benefit to keeping the page size information only in the PTEs. It *allows different processes to map the same large page with different page sizes*. For example, suppose two processes memory map the same file into their address space. In this case they have to share the same set of physical pages. Let us also assume that one process maps the first 64K bytes of the file while the other maps the first 48K bytes. The first process may choose to map the file into a 64K page while the other process may choose to use three 16K pages. Both processes share the same physical pages but map them with different sizes. The page size choices one process makes do not affect others sharing the same file. Likewise, downgrading this virtual address range for the first process is completely transparent to the second process. This is possible because the page size information is only kept in the PTEs and hence private to a process.

This approach is not without its disadvantages. It does not reduce the size of the page tables required if one uses large pages. All the attributes and permissions in the set of PTEs that encompass the large page have to be kept consistent, although the number of cases where we need to maintain this consistency is limited.

6.3. Software TLB Miss Handler

On MIPS processors, a TLB miss generates an exception. The exception is handled by a TLB miss handler. It traverses the page table for the given virtual address and drops in the contents of the PTE into the TLB. The TLB miss handler has to be very efficient and is usually only a few instructions long. Otherwise the time spent handling TLB miss exceptions can be a significant fraction of the total runtime of the application. The format of the PTE directly affects the performance of the TLB miss handler. The more the PTE looks like a TLB entry, the faster the TLB miss handler can drop the entry into the TLB. As explained below, a TLB miss handler that supports multiple page sizes has more overhead compared to a single page size TLB miss handler because the MIPS processor has a separate page mask register [Mips94]. On a single page size system, the page mask register is set once at system initialization time to the fixed page size. On such systems, the TLB miss handler does not modify the page mask register. To support multiple page sizes, the TLB miss handler

has to set the page mask register in addition to other entries during each TLB miss. So the single page size TLB miss handler is much cheaper compared to the multiple page size TLB miss handler. This means that if we were to use the multiple page size TLB miss handler for all processes in the system, then processes that did not use large pages would have a performance impact. To avoid this problem we use a feature of IRIX that allows us to ***configure a TLB miss handler per-process***. Thus only processes that need large pages use the multiple page size TLB miss handler. All other processes use the fast single page size TLB miss handler. This supports one of our main goals: that processes not using the large page feature are not burdened with additional overhead.

6.4. Large Page allocator

This section discusses issues related to the allocation of large pages. It describes a mechanism called page migration to unfragment memory and also describes a kernel thread called the coalescing daemon that uses page migration to coalesce large pages.

6.4.1. Page allocation issues

Traditionally in single page size systems, pfdats of free pages are kept in linked lists called *free lists*. Allocating and freeing pages is done by simple removal and insertion of pfdats into the free lists. To be able to allocate pages of different sizes at runtime, the physical memory manager should manage memory in variable size chunks and handle external fragmentation. In busy systems, the pattern of allocating and freeing pages causes the free memory to be so fragmented that it is difficult to find a set of contiguous free pages that can be coalesced to form a large page. This problem of allocating chunks of different sizes has been very well studied in literature. The problem can be divided into two parts. One is to minimize fragmentation while allocating pages and the other is to provide mechanisms to unfragment memory.

We have designed an algorithm that keeps the overhead of the allocation and freeing procedures to a minimum and leaves the work of unfragmenting memory to a background kernel thread. This allows better control of the physical memory manager. For example, if an administrator chooses not to configure large pages, the background thread will not run and there will not be any additional overhead compared to a system with one base page size. Alternatively, the administrator can configure large pages and make the background thread aggressively unfragment memory.

To minimize fragmentation, the manager keeps free pages of different sizes on different free lists, one for each size. The allocation algorithm first tries to allocate pages from the free list for the requested page size. If a free page cannot be found, the algorithm tries to split a page of the next higher size.

The manager also uses a *bitmap* to keep track of free pages. For every base sized page in the system, there is a bit in the bitmap. The bit is set if the page is free and is cleared if the page is not free. This helps in coalescing a set of adjacent pages to form a large page as we can determine if they are free by scanning the bitmap for a sequence of bits that are set. When a page is freed, the bitmap is quickly scanned to see if a large page can be formed and if so pages corresponding to the bits in the bitmaps are removed from their free lists and the first pfdat of the newly formed large page is inserted into the large page free list. Note that the manager must also ensure that the processor alignment restrictions are met (the physical address of the large page should be aligned to the page size boundary) when coalescing the pages.

This algorithm uses high watermarks to limit the coalescing activity. The watermarks provide a high degree of control over the allocator and can be changed by the system administrator, even while the system is running. The high watermarks provide upper limits on the number of free pages of a given page size. Coalescing activity stops once the high watermarks have been reached.

6.4.2. Page Migration

As mentioned earlier, on a long running system a mechanism to *unfragment* memory is needed, since pages can be randomly allocated to the kernel, the file system buffer cache and to user processes. It is very difficult in these cases to find a set of adjacent free pages to form a large page even though there is a lot of free memory left in the system. To solve this problem we use a mechanism called ***page migration***. Page migration transfers the identity and contents of a physical page to another and can be used to create enough adjacent free pages to form a large page.

For example, in Figure 3, assume a 16K chunk of memory (A) has four adjacent 4K pages, three of which are free and one (A4) is allocated to a process. By transferring the contents of the busy page A4 to a free page B1 from another chunk B, we can free chunk A completely and thus use it to form a 16K large page.

The page migration algorithm that replaces page A4 with page B1 works as follows.

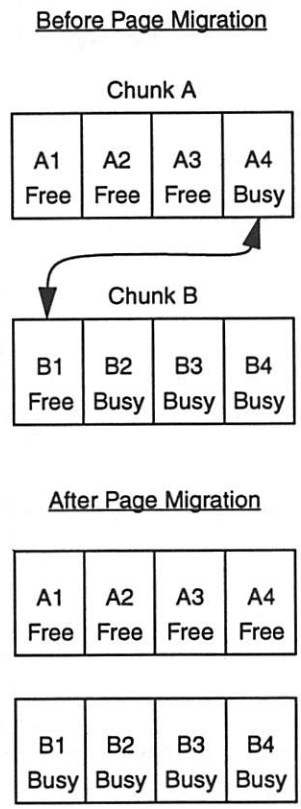


Figure 3: Page Migration Example.

a) It checks to see if the page A4 can be replaced. Pages that are held by the kernel for DMA or other purposes cannot be replaced. Pages that are locked in memory using the `mlock()` system call are also not replaced.

b) The next step is to get rid of any file system buffer references to page A4.

c) The page tables entries that map the page A4 are modified to point to page B1. The PTE valid bits are turned off and a TLB flush is done to force the processes that map page A4 to take a page fault. The pfdat of page B1 is marked to indicate that its data is in transit. This forces the processes to wait for page B1 to be ready. PTEs that map page A4 are found using its *reverse map* structure for page A4. The reverse map structure is part of the pfdat and contains a list of PTEs that map that page. At the end, the reverse map structure itself is transferred to page B1.

d) Page A4 is removed from the page cache and page B1 is inserted in its place.

e) Page A4's contents are copied to page B1. Page B1 is marked ready and the processes (if any) that were sleeping waiting for page B1 to be ready are woken up.

In practice a set of pages are migrated together to minimize the number of global TLB flushes.

6.4.3. Coalescing Daemon

The mechanisms to unfragment memory is implemented by a background thread called the ***coalesce daemon***. The daemon scans memory chunks of different sizes. It goes through several passes each with a different *level of aggressiveness*. There are three levels. In the first level or the *weak* level, it does not do page migration. The daemon cycles through all the pages looking for free contiguous pages and tries to coalesce them into a large page. In the second level or the *mild* level, it limits the number of page migrations per chunk. The daemon examines each chunk and computes the number of page migrations needed to make that chunk into a large page. The number of page migrations should be equal to the number of clear bits in the bitmap for that chunk. If the number is below a *threshold* the daemon proceeds to migrate that page. There is a threshold for every page size. The thresholds have been chosen by experimentation. The third or the *strong* level uses page migration very aggressively. For every chunk, the daemon tries to migrate all the pages that are not free. For most of the passes the daemon uses the first two levels. If after several passes the daemon is still not able to meet the high watermark it uses the strong level.

If a process wants to allocate a large page but the request cannot be satisfied immediately, it can choose to wait (by setting a special policy as described in the next section). If a process is waiting for a large page, the coalesce daemon uses the strong level of aggressiveness. This allows the daemon to quickly respond to the process's large page needs.

6.4.4. Minimizing Fragmentation

There are limitations to the page migration technique. Pages that have been allocated to the kernel or have been locked in memory cannot be migrated. The reason is that pages allocated to the kernel are generally not mapped via page tables and the TLB. For example, IRIX kernel memory is directly mapped through the K0SEG segment of the virtual address space of the R10000. Such pages cannot be migrated since many kernel subsystems have direct pointers to such pages. Pages that have been locked for DMA or locked using the `mlock(2)` system call cannot be migrated either,

since the callers assume that the underlying physical page does not change. Consequently the coalescing daemon performs well if most chunks contain pages that can be migrated. This is the case with many supercomputing applications which have large data segments whose pages can be migrated easily. In such systems the percentage of memory taken up by the kernel is very low and hence most of the memory is migratable. This is not true for desktop workstations running graphics applications. They usually have a limited amount of physical memory and a significant percentage of it is used by the kernel or locked for doing DMA. For these machines, we have extended the physical memory allocator to keep track of migratable pages. The physical memory manager is notified at the time of page allocation whether the page will be migratable. The manager keeps track of the number of migratable pages per chunk. If the manager wants to allocate a page for the kernel, it chooses a page from the chunk which has the least number of migratable pages. On the other hand, if the manager wants to allocate a migratable page it chooses a page from the chunk with the most number of migratable pages. This algorithm has the advantage of maximizing the number of chunks whose pages can be migrated although it adds overhead to the page allocating and freeing algorithms by keeping track of a list of chunks sorted by the number of migratable pages per chunk. This algorithm is used in low end workstations, where the advantage of being able to allocate large pages under conditions of severe fragmentation outweigh the disadvantage of the overhead incurred while allocating and freeing pages.

6.5. Policies Governing Page Sizes

IRIX provides a facility by which application can choose specific VM policies to govern each part of its address space. It provides system calls [IRIX96] to create policies and attach them to a virtual address range of a given process. The `pm_create(2)` system call allows an application or a library to create a policy module descriptor. The system call takes several arguments which specify the policy as well as some parameters needed by the policy. The `pm_attach(2)` system call allows an application to attach a policy module descriptor to a virtual address range.

The application can choose among a variety of policies provided by the kernel. Policies fall into several categories. The *page allocation* policies specify how a physical page should be allocated for a given virtual address. For example on a NUMA system, the application can specify that pages for a given virtual address range should be allocated from a node passed as a parameter to the policy. Another policy decides whether allocating a page of the right cache color takes precedence over allocating a page from the closest node. The *migration policies* allow the application to specify whether the pages in a given virtual address range can be migrated to another NUMA node if there are too many remote references to that page. It can also be used to set the threshold that triggers the migration.

Page size hints for a given virtual range can be specified via a policy parameter. `pm_create()` takes the page size as an argument. Thus the application or a runtime library can specify which page size to use for a given virtual address range. The kernel tries to allocate a page of the specified page size as described in the next section. The page size parameter works in conjunction with other page allocation policies. For example, the page allocation policy can specify that a page has to be allocated from a specific NUMA node and that it should be a large page. There are currently two large page specific policies:

a) On NUMA systems, if large pages are not available on an application's home node, the kernel can either borrow large pages from adjacent nodes or use lower sized pages on the home node. By default the kernel borrows a large page from an adjacent node but the application can indicate that locality is more important using this policy.

b) If a large page of the requested size is not available, the application can wait for the coalesce daemon to run and coalesce a large page. Sometimes the wait time is not acceptable and in that case the application can choose to use a lower page size (even the base page size). The application can specify if it wants to wait for a large page and also the time period for which it wants to wait before using a lower page size. The coalescing daemon runs aggressively when a process is waiting for large pages.

These policies will be refined and new ones will be added as we learn more about applications and how they behave with large pages. In the future, we want to be able to automatically detect TLB miss rates using the performance counters provided by the processor and upgrade a virtual address range of a process to a specific page size. We decided not to do on-line page size upgrade initially as it was not clear the performance benefits would outweigh the cost of collecting the TLB miss data and the page size upgrade. The kernel always uses the base page size unless the application or a runtime library overrides the default.

Currently the page size hints are provided by the application or special runtime libraries like the parallelizing Fortran compiler runtime, MPI runtime, etc. These runtime libraries usually have a better understanding of the application's behavior and can monitor the performance of the application.

6.5.1. Tools to Specify Page Size Hints Without Modifying Binaries

Applications need not be modified or recompiled to be able to use large pages. IRIX has a tool called `dplace(1)` [IRIX96] that can be used to specify policies for a given application without modifying the application. The tool inserts a dynamic library that gets invoked before the application starts. The library reads configuration files or environment variables and uses them to set up the page size and NUMA placement policies for a given virtual address range. For example the following command,

```
dplace -data_pagesize 64k ./a.out
```

sets the policies for the address space of a.out such that its data section uses 64K page size.

IRIX provides a valuable tool called `perfex(1)` [Marco96] that allows a user to measure the number of TLB misses incurred by a process. It gets the data from the performance counters built into the R10000 processor.

Another tool called `dprof(1)` analyzes memory reference patterns of a program and can be used to determine which parts of an application's address space encounters the most TLB misses and hence will benefit from large pages.

The parallelizing fortran compiler runtime also takes page size hints via environment variables and can be used to set up page size hints for applications that have been compiled with that library.

6.6. Page Faulting

The page fault handler is invoked whenever a reference is made to a virtual address for which the PTE is not marked as valid. The page fault handler allocates and initializes a page and sets up the PTE for the virtual address that generated the TLB miss. Initializing the page can involve finding the page in the page cache for the memory object, reading in the data from a disk file or a swap device, or just zeroing the page. Page faults usually happen when a process accesses the virtual address for the first time as the page tables will not yet be initialized. The fault handler's functionality must be extended to handle large pages. In particular, the handler should be able to allocate and initialize large pages, ensure that the processor restrictions are not violated and set up the PTEs.

Adding these extensions to the handler is simplified by our choice of data structures for the pfdats and the PTEs. The extensions are minimal and the majority of the fault handler remains the same as the base page size fault handler. The large page fault handler is written as a layer on top of the standard page fault handler for base pages. The fault handler is described below. Let us assume that the virtual address is `va`.

a) The first step is to consult the policy module (described above) to determine the page size that should be mapped. The virtual address is then aligned to this page size. Let us call the aligned virtual address `ava`.

b) The next step is to verify that the virtual address range `[ava, ava + page_size)` has the same protections and other attributes. As discussed earlier, since there is only one TLB entry per large page, the protections and the attributes have to be the same for the entire virtual address range mapped by the TLB entry. It also verifies that no pages of a different size are already present in the range `[ava, ava + page_size)`.

c) The handler verifies that both sub-entries in a TLB entry that maps this address range have the same page size. This satisfies the MIPS R10000 processor restriction.

d) The handler then allocates a large page. If a large page cannot be allocated and the process has chosen to wait (specified via a policy) for a large page, it waits for a large page. If a large page of the required page size can be obtained it retries the algorithm. If after a timeout period it cannot get a large page or the process chose not to wait, the handler retries the algorithm with a lower page size.

e) The handler enters a loop and faults each base page in the large page one at a time. Thus for a 64K page, the handler has to loop 16 times. The fault algorithm is identical to the base page fault algorithm. This is where the page is initialized, read from a file, swapped in from disk, etc. We can do this because the large page pfdats and PTEs look identical to those of base page pfdats and PTEs.

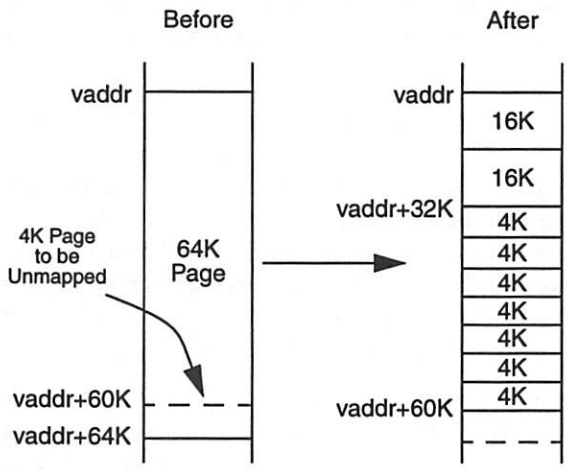


Figure 4: Downgrading a 64K Page.

f) Once the pages have been faulted, the page size field is set in the PTEs and they are made valid. Note that a large page is mapped by many adjacent PTEs. The handler also drops in a TLB entry for `va`.

6.7. Downgrade Mechanism

The motivation behind downgrading comes from system calls whose behavior is highly tied to the base page size. UNIX system calls like `mmap()`, `munmap()` and `mprotect()` work on virtual address ranges at the granularity of the base page size. If we map a large page to a virtual address range and the application makes a system call to change the protections for part of the address range, we can either return a failure to the system call or downgrade the large page to a lower size small enough to enforce the protection criteria. Downgrading allows the operating system to not change the system API and ABI, while internally using different page sizes. This is a huge advantage as it allows applications to run unmodified.

The downgrade algorithm allows us to have no restrictions on which parts of the address space can be mapped with large pages. For example, there is no restriction that one region or segment of the address space should be of the same page size. Pages of different sizes can be stacked one on top of the other.

Figure 4 shows an example where a large page is downgraded automatically by the kernel to satisfy an `munmap()` system call. The virtual address range `[vaddr, vaddr+64K)` has been mapped with a 64K page. Suppose the application wants to unmap the last 4K bytes of that address range (as indicated in Figure 4) using the `munmap()` system call. In this case, we have to downgrade the 64K page into smaller page sizes. The downgrade algorithm also has to obey the R10000 processor restriction that page sizes of the pages at the even and odd address boundaries must be the same. For example, since `vaddr` is guaranteed to be aligned at a 64K boundary, `vaddr` is also aligned at an even 16K boundary. The downgrade algorithm downgrades the page size of the PTEs mapping the range `[vaddr, vaddr+32K)` to 16K. The address range `[vaddr+32K, vaddr+64K-16K)` cannot be mapped by two 16K pages as the range is not big enough. Nor can a single 16K page be used in the range `[vaddr+32K, vaddr+32K+16K)` because of the hardware restriction that even and odd pages must be the same size. So the algorithm downgrades the range to the next lower page size (4K) which happens to the base page size.

The downgrade algorithm is quite simple. It first clears the valid bit of all the PTEs that map the large page and then flushes the TLB. The TLB flush only flushes those TLB entries which map the given virtual address range of the process. The invalidation is necessary to make the update of the page size atomic. Once the TLB has been flushed, the page size fields in all the PTEs are updated to reflect the new page size. The new page size is the next lower page size to which the address range can be downgraded without violating the MIPS processor restrictions.

In practice, downgrades rarely happen for large supercomputing and database applications, as they do not spend much time mapping and unmapping their address spaces.

6.8. Upgrade Mechanism

An upgrading mechanism is useful to dynamically increase the page size of an application based on feedback from other parts of the operating system. Processors like the MIPS R10000, provide counters that track the TLB misses incurred by a process. This counter can provide feedback to the VM manager to upgrade the page size. As said earlier, we have decided not to do on-the-fly page upgrade based on feedback from the process's TLB miss profile. Currently an application or the runtime can advise the operating system to upgrade the page size for a given virtual address range that corresponds to a key data structure using a new command to the `madvise()` system call.

The system call is particularly useful for upgrading text pages. Consider the case where an executable is invoked and the base page size is used. The text file cor-

responding to the executable is faulted into base pages and the pages become part of the file's page cache. If the same executable is now run a second time with large page size hints, it is quite unlikely that the kernel will be able to use large pages as the pages for the text are already in the page cache and hence it has to reuse them. In this situation the application can invoke the `madvise()` system call to upgrade its existing text pages to the large page size. The algorithm to upgrade a page from its old size to a new size is described below.

a) The first step clears the valid bit for the PTEs that span the large page and performs a TLB flush.

b) Next a large page of the requested size is allocated.

c) We use the page migration algorithm (described in section 6.4.2) to replace the old pages in the virtual address range with the large page.

d) The PTE size fields are updated to reflect the new size and they are validated.

6.9. Page Replacement Algorithm Extensions

Most page replacement algorithms (like the BSD clock algorithm [BSD89]) usually use reference bits in the PTEs to keep track of recently referenced pages for a given process. Pages which are not recently referenced are usually paged out to disk. The memory reference patterns are tracked for the entire large page. The default paging policy for large pages is simple. If part of a large page is chosen to be swapped out we downgrade the large page. We try to minimize downgrades and improve performance by swapping out entire large pages.

7 Performance

Figures 5 and 6 show the performance improvements obtained when using large pages on some standard benchmarks. The benchmarks were performed on an SGI Origin 2000 running IRIX 6.4. The base page size for that system is 16K. The benchmarks were run with 16K, 64K, 256K and 1M page sizes. *turb3d, vortex* and *tomcatv* are from the Spec 95 suite [Spec95]. *Fftpde* and *appsp* are from the NAS parallel suite. The benchmarks were not modified. All of them were single threaded runs. The same binaries were used for all the runs. Figure 5 shows the percentage reduction in the number of TLB misses and Figure 6 shows the percentage performance improvement at different page sizes

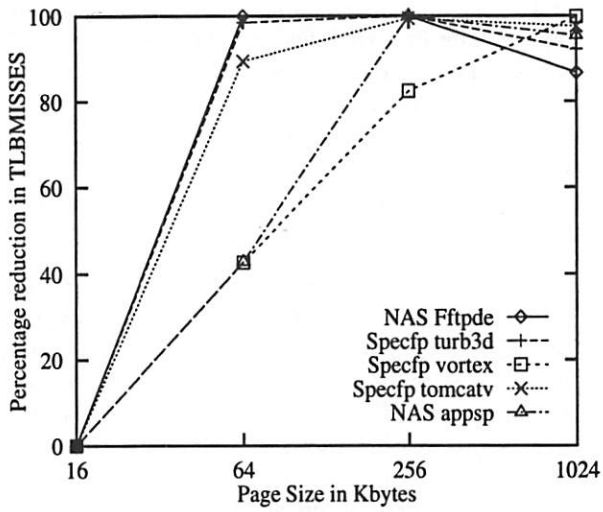


Figure 5: Percentage Reduction in TLB Misses with Large Pages.

with respect to the base page size (16K). The page size hints were given using `dplace(1)`. From Figure 6 we can see that some applications get 10 to 20% improvement with large pages. The improvements will be even greater for larger problem sizes.

The *TLB miss overhead* (defined as the ratio of the time spent handling TLB misses to the total runtime of the application) is a significant part of the runtime for *vortex, turb3d* and *appsp*. The overhead is minor for *tomcatv*. We can deduce this by looking at Figures 5

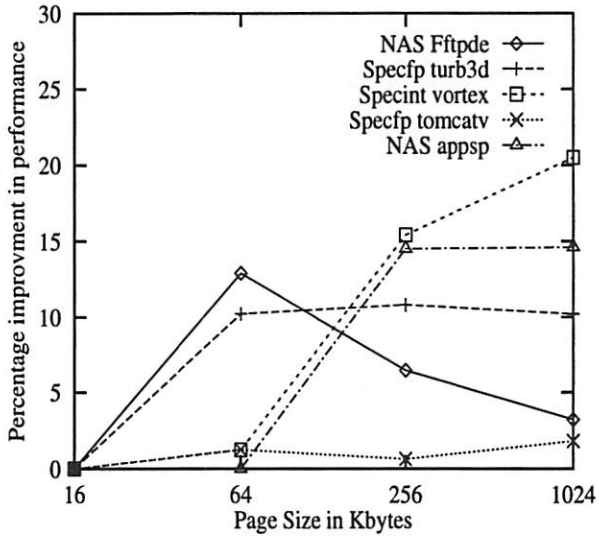


Figure 6: Percentage Improvement in Performance with Large Pages.

and 6. We can see that for *turb3d* and *vortex*, the performance improves significantly with reduction in TLB misses. For example, vortex gives a 20% improvement in performance when using 1MB pages. On the other hand although *tomcatv*'s TLB misses were significantly reduced with 64K pages (Figure 5), the performance improvement (Figure 6) is negligible for the same page size.

Once the TLB miss overhead has been reduced by about 98%, further increases in page sizes provide diminishing returns. For example, *turb3d* does not show much improvement beyond 64K pages. Some times at higher page sizes, the operating system cannot map large pages for the entire address space range due to alignment restrictions. So it may be forced to use base pages. This causes the performance to drop due to higher TLB misses. We can see this behavior with *fftpde* when using 1MB pages.

Figure 7 shows how large pages perform under contention. The experiment measures the performance of the *fftpde* NAS benchmark using 64K pages on a single CPU Origin 200 system with a variety of threads running in the background. Three kinds of background threads were used to simulate different TLB and cache usages. In one case the background thread is a simple spinner program. The program is an extremely small tight loop that is completely CPU bound and fits into the CPU's cache and TLB and should not cause too many TLB misses. In the second case, the background thread is another invocation of the *fftpde* program using 16K pages. This should cause considerable thrashing of the TLB and the cache. The third case also uses *fftpde* program as the background thread but this time with 64K pages. This should reduce the TLB thrashing as the working set can fit into fewer TLB entries. The performance improvement was measured with an increasing number of background threads. As we can see, the performance degrades with contention but the drop is not so steep if the background thread is a spinner or a *fftpde* program with 64K pages. When the background thread is an *fftpde* program with 16K pages, more TLB entries are replaced by the background threads and hence the performance degradation is more. The benefits pro-

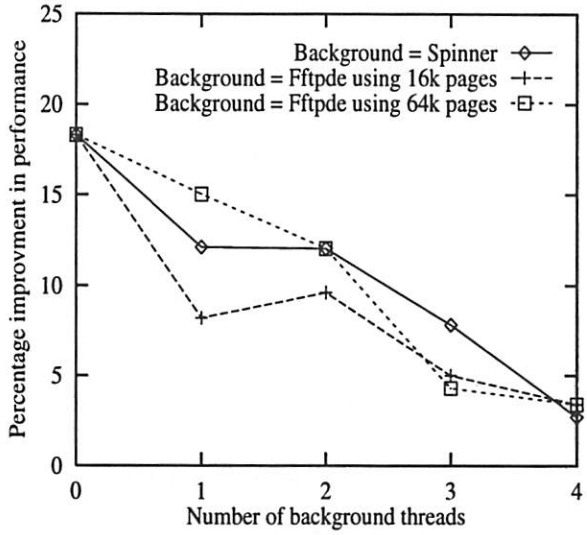
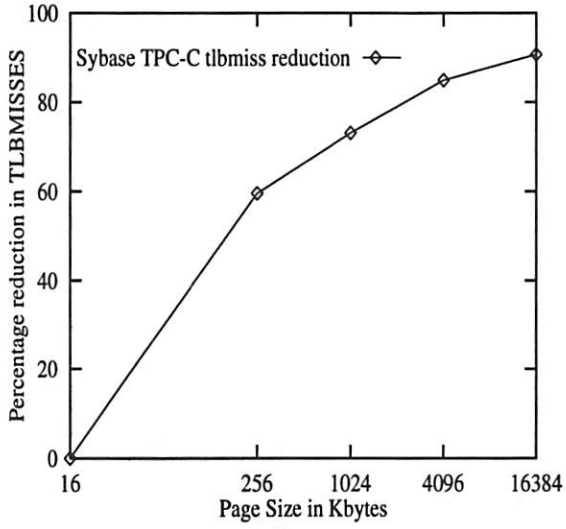


Figure 7: Large Page Size Performance Improvement under Contention.



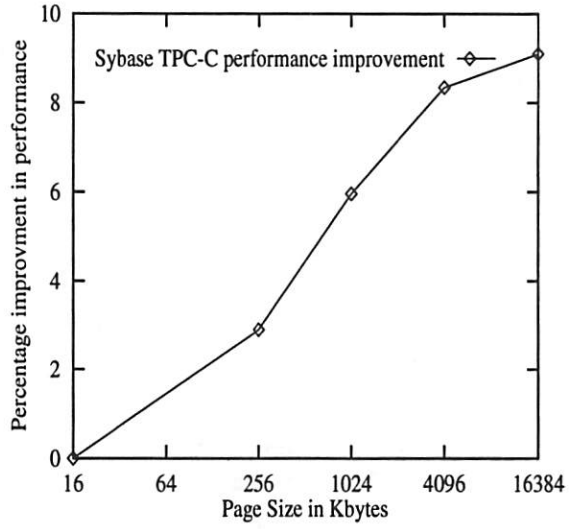


Figure 8: Large Page Size Performance Improvement for a Commercial Benchmark.

vided by large pages decrease with increase in number of background threads as fewer TLB entries are available and more TLB entries have to be replaced.

Figure 8 shows an example of how large pages can benefit a real business application. It shows the percentage reduction in TLB misses and the percentage improvement in performance at various page sizes for the TPC-C benchmark run on a two processor Origin 200QC using the Sybase Adaptive Server Enterprise database program. The program has very poor locality of reference and very large working set and consequently suffers from a high TLB miss overhead. As seen from the graph, we need very large pages (16M) to almost completely eliminate the TLB miss overhead.

8 Future Directions

One of our plans is to investigate the performance benefits of providing on-line page size upgrade based on feedback from the R10000 TLB miss counters. We also have opportunities to improve the performance of the large page allocator by minimizing page fragmentation. Some new policies may be added based on feedback from users who have used large pages on commercial applications.

9 Acknowledgments

We would like to thank Paul Mielke, Luis Stevens and Bhanu Subramanya for the many discussions we had during the course of this project. We also thank William Earl, Brad Smith and Scott Long for providing insight on the mechanisms to minimize fragmentation on workstations. We thank Marco Zhaga for providing us with performance data for a commercial database application.

10 References

[BSD89] Samuel Leffler, Kirk McKusick, Michael Karels, John Quarterman, The Design and Implementation of the 4.3BSD Unix Operating System, *Addison-Wesley, ISBN 0-201-06196-1*, 1989.

[Cox94] Berny Goodheart, James Cox. The Magic Garden Explained: The Internals of Unix System V Release 4, *Prentice Hall, ISBN: 0132075563*, 1994.

[Denn70] Peter J. Denning. Virtual Memory, *Computing Surveys, 2(3):153-189,* September 1970.

[IRIX96] IRIX 6.4 man pages for `dplace(1)`, `dprof(1)`, `pm(3)`, `madvise(2)`, *Silicon Graphics,* 1996.

[Khal93] Yousef Khalidi, Madhusudhan Talluri, Michael N. Nelson, Dock Williams. Virtual memory support for multiple page sizes. In *Proc. of the Fourth Workshop on Workstation Operating Systems,* pages 104-109, October 1993.

[Marco96] Marco Zhaga, Brond Larson, Steve Turner and Marty Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of Supercomputing '96, Nov '96.*

[Mips94] Joe Heinrich. MIPS R10000 Microprocessor User's Manual, *MIPS Technologies, Inc.*, 1994.

[PA-RISC] Ruby Lee and Jerry Huck, 64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture. On-line documentation. *http://www.hp.com/wsg/strategies/pa2go3/pa2go3.html.*

[Romer95] Theodore H.Romer, Wayne H.Ohlrich, Anna R.Karlin and Brian N. Bershad. Reducing TLB and Memory overhead Using On-line Superpage Promotion. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[Schim94] Curt Schimmel, UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers, *Addison-Wesley, ISBN 0-201-63338-8*, 1994.

[Spec95] SPEC news letter. On-line documentation at *http://www.spec.org/osg/news/articles/news9509/cpu95descr.html,* September, 1995.

[Sparc97] Ultrasparc II data sheet. On-line documentation at *http://www.sun.com/microelectronics/datasheets/stp1031/.*

[Tall94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171-182, October, 1994.

[Tall95] Madhusudhan Talluri, Mark D. Hill and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proc. of Symposium of Operating System Principles (SOSP),* Dec 1995.

Implementation of Multiple Pagesize Support in HP-UX

Indira Subramanian
Cliff Mather
Kurt Peterson
Balakrishna Raghunath

Hewlett-Packard Company
Cupertino, CA 95014
indira@cup.hp.com

Abstract

To reduce performance degradation from Translation Lookaside Buffer (TLB) misses without significant increase in TLB size, most modern processors implement TLBs that support multiple pagesizes. For example, Hewlett-Packard's PA-8000 processor allows 8 hardware pagesizes, in multiples of four, ranging from 4 Kbytes to 64 Mbytes.

In implementing multiple pagesize support in HP-UX, we chose to create large pages at page-fault service time. We have a buddy system allocator that provides interfaces for allocating and freeing multiple pagesizes. We maintain the Virtual Memory (VM) data structures such as the pagetable entry, virtual page frame descriptor, and physical page frame descriptor based on the smallest pagesize, and represent a large pagesize as a collection of these base pagesize structures. In our implementation, VM operations on a large pagesize such as 16KB are carried out by looping over the 4KB-based constituent VM data structures. Our system offers significant application performance improvement when using large pagesizes.

1 Introduction

Translation Lookaside Buffer (TLB) misses can degrade the performance of applications with large working set sizes [2, 4, 18, 21, 23]. A TLB is a cache of recently accessed virtual-to-physical page translation information. The *working set* of a process is the memory actively referenced during a certain time interval [6]. A typical TLB that performs translations using small pagesizes such as 4KB, cannot hold all the translations for a large working set. Consequently, TLB misses will result, and each miss must be handled by copying the translation information from a software or a hardware translation table (pagetable) into the TLB. A high TLB miss rate (misses per second) will result in performance degradation. While this degradation is common to all contemporary processors, the penalty can be significant in processor implementations that lack hardware support for TLB miss handling.

To increase the TLB reach, that is, the amount of memory translated by the TLB, most modern processors support multiple pagesizes. Support for a wide range of pagesizes allows for miss reduction without undue increase in working set sizes. For example, in addition to the base 4KB pagesize, the PA-8000 processor which is an implementation of the PA-RISC 2.0 architecture [10], supports 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, and 64MB pagesizes. Other architectures that allow multiple pagesizes include DEC Alpha [3], MIPS R10000 [16], and SPARC [7].

Several issues need to be considered when implementing OS support for multiple pagesizes [11, 17, 21]. How should the VM data structures, which were originally designed to represent a uniform pagesize such as 4KB, be redesigned or adapted to allow coexistence of multiple pagesizes? How will the pagesize be chosen for a given mapping? Should a large mapping be created at fault service time? Or, should it be created at a later time through page promotion [18, 23]? How should the interfaces across VM and filesystems be modified to deal with multiple pagesizes? How should physical memory be managed such that a properly aligned page of any TLB-supported pagesize may be allocated? How should candidates for page replacement be selected?

To support multiple pagesizes in HP-UX, we preserve the underlying 4KB-based VM data structures, and represent a large page by a collection of these 4KB-based structures. In this approach, a large page

is effectively a set of contiguous 4KB sized pages. The PA-RISC 2.0 TLB requires that the virtual and physical addresses of a large page be aligned on the pagesize boundary. We use the terms *subpage*, *base page*, and *member page* interchangeably to refer to the 4KB pages that constitute a large page. The VM subsystem components such as the fault path, the interfaces that support page fault handlers, and the physical memory allocator operate on a large page by looping over the associated base page data structures. Our implementation offers significant performance improvement for applications with large working set sizes.

The remainder of this paper is organized as follows. Section 2 discusses related work. In Section 3, we describe the HP-UX VM system primarily focusing on the data structures, and discuss the rationale for our approach to representing large pagesizes. Section 4 describes the pagetable management. Section 5 presents the physical memory allocator. Section 6 discusses how pagesize hints may be specified for an application. The VM subsystem uses this pagesize hint as the starting point, when deciding the pagesize for mapping an address range. In Section 7, we describe support for large page creation at fault-service time. Section 8 discusses page replacement in the presence of multiple pagesizes. Section 9 presents performance data demonstrating performance improvements from TLB miss reduction. In Section 10, we summarize our approach to multiple pagesize support and reaffirm its benefits.

2 Related Work

Several approaches for supporting multiple pagesizes have been reported in the literature. In this section, we describe these approaches and relate them to our work.

2.1 Pagesizes in Partitioned Memory

Large pagesize and adaptive prepaging are among the techniques Kagimasa et al. employ to reduce memory management overhead in a terabyte virtual and gigabyte physical memory system [9]. The goal of their Super Terabyte System (STS) was to reduce overhead from page fault processing, choosing candidates for page replacement, and process swapping. Kagimasa et al. do not discuss performance effects of TLB misses. Nonetheless, their approach for implementing dual pagesize support is relevant to our discussion.

STS supports one small pagesize (4KB) and one large pagesize simultaneously. The large pagesize is one of 16KB, 64KB, 256KB, and 1MB, and is chosen at boot time. The virtual and the physical memory are each partitioned into a small page region and a large page region. The size of the virtual small page region is 2 gigabytes, while that of the physical small page region can be set at boot time. A system available physical page area (SAPA) is maintained for each of the two pagesizes. When setting up a mapping for a large virtual page, in the event the SAPA for large pages is empty, a contiguous set of small physical pages are located if possible, and used. Similarly, when setting up a mapping for a small virtual page, if the SAPA for small pages is empty, a small page is carved out of a large physical page, and the remaining small pages are placed into the local available page area (LAPA) for the process.

The system manages the two pagesizes as follows. The key storage (per physical page data structures) is maintained for each 4KB page. Large pagesize is used for allocation, page fault handling, setting referenced and modified bits, page replacement, and swapping. However, management of secondary storage as well as reading and writing operations involving secondary storage utilize 4KB pagesize.

With regard to support for multiple pagesizes, the STS design offers some flexibility but suffers from several drawbacks. The 4KB-based physical page data structures facilitate the allocation of small and large physical pages interchangeably. This flexibility enables efficient use of available physical memory in the two pagesizes. There is one limitation however – all the small pages carved out from a large page must be mapped to the same process. STS has two major drawbacks. First, only two pagesizes are supported. The large pagesize chosen at boot-time may not be appropriate for a broad range of applications. Second, virtual storage is statically partitioned into small page and large page regions. Such static partitioning may not be best suited for TLB miss reduction for different workloads.

2.2 Subblocked TLB

Talluri et al. recommend subblocked TLB organizations as better alternatives to existing TLB organizations that have been simply extended to support multiple pagesizes [21]. *Subblocking* refers to grouping of mapping information in the TLB for several base pages that are part of a page block. A *page block* is made up of 16 4KB pages that are aligned on a 64KB boundary. Subblocking saves TLB space by sharing the virtual tag across all the subpages of the page block. A *complete-subblock* TLB entry stores protection and other page attributes and a physical page-number for each of the subpages in the page block. A

partial-subblock TLB entry stores a single set of page attributes and a single physical page-number for the entire set of subpages in the page block, and therefore requires less TLB space.

Talluri et al. propose subblock TLB designs as an alternative to multiple pagesizes for improving TLB performance. They argue that invasive changes to the OS that introduce significant overhead are necessary to take advantage of multiple pagesizes. Overheads include increased disk and network traffic for page-ins and page-outs, coalescing smaller pages to create large pagesizes, and existing VM data structures not scaling efficiently for handling large pagesizes. In contrast, the complete-subblocking approach requires no changes and the partial-subblocking approach requires minimal changes to the OS for improving TLB performance. To exploit the benefits from partial-subblocked TLB, as many subpages as possible in a virtual page block must be mapped to the corresponding subpages in a physical page block. For this purpose, a reservation based memory allocation is used, and is discussed in the next section.

Our goal was to improve the TLB performance of processors based on the PA-RISC 2.0 architecture, which supports multiple pagesizes. Our implementation demonstrates that multiple pagesizes can be exploited effectively to improve TLB performance.

2.3 Page Reservation and Promotion

In the case of partial-subblocking (discussed above) and multiple pagesize TLB, Talluri et al. employ a reservation-based allocation of a page block [20, 21]. Reservation refers to setting aside properly aligned 4KB physical pages for possible use with specific virtual subpages of a process. These physical subpages are placed at the end of the freelist. When the process references these virtual pages, the ensuing page faults are serviced using the prereserved physical subpages. If the process did not reference these virtual pages, some of the reserved pages may move to the head of the freelist. These unused reserved pages are then allocated to service other page faults. A *single-page-size framework* is employed to support two page sizes – a 64KB page is represented by 16 4KB page-based data structures.

In addition to the reservation method described above, for supporting two pagesizes, their system implements a threshold-based promotion policy. This policy decides when to combine the 4KB subpages to create a 64KB superpage (large page). After a certain *promotion threshold* such as 50% of 4KB pages have been faulted in, the unreferenced pages in the page block are fetched from secondary storage into corresponding prereserved 4KB subpages. The page block is then promoted to 64KB pagesize. In the case of uninitialized data, promotion involves only zerofilling the prereserved pages. If some of the prereserved pages are no longer available, a *gather* operation may be needed when performing page promotion. In this case, a new page block is allocated, and the original 4KB physical pages are copied into the new page block to create a large page mapping. Talluri did not implement this gather mechanism in his system[20].

Talluri's system poses several limitations. First, it provides simultaneous support for only 2 page sizes, 4KB and 64KB. We use the single-page-size framework as well in our implementation, and we demonstrate that this method is suited for all pagesizes that the PA-8000 processor supports. Second, it is unclear that the reservation approach will scale well for larger pagesizes. A promotion threshold of 50% would involve many faults before the large page is created by promoting the subpages. Furthermore, if some of the prereserved pages were unavailable resulting in random physical subpage allocations, promotion will require copying the source subpages. In contrast, we allocate and map large pages when servicing a page fault, thereby eliminating the additional faults and the promotion overhead.

2.4 Clustered Pagetable

Through simulation, using estimates of pagetable size and access time as metrics, Talluri et al. [22] demonstrate that clustered pagetables work better for superpages (large pages) than the conventional hashed pagetables. A *hashed* pagetable organization [8] uses a hash function that hashes a virtual page number to a specific hash bucket in which the translation information for the virtual page is stored. A *clustered* pagetable is a hashed pagetable enhanced with subblocking. *Subblocking* refers to grouping of mapping information for several pages, and it amortizes the per pagetable entry (PTE) overhead over many potential mappings. The aligned group of consecutive pages is called a page block. Space saving is achieved by using a single virtual tag and a single hash chain pointer for the entire subblock. In a clustered pagetable with a subblocking factor of 16 and a base pagesize of 4KB, a single clustered PTE can support pagesizes up to 64KB. A clustered pagetable provides effective support for a subblocked TLB, which was discussed earlier in Section 2.2.

Talluri et al. present approaches for storing a large page in different types of pagetables. Two solutions that work well are the multiple pagetables method and the replicated PTE method. The multiple pagetable

method entails one pagetable for each of the pagesizes used in the system. To locate the mapping that caused a TLB miss, the TLB miss handler must search each of the pagetables, starting from the most likely table. A more promising approach is the *replicated* PTE method, in which a large page is represented by replicating a PTE once for each subpage.

With a clustered pagetable design, representation of pagesizes larger than page block size involves a space/time tradeoff as in conventional tables but are more efficient. Assuming a subblocking factor of 16 and a base pagesize of 4KB, pages larger than a page block (64KB) can be represented by replicating the 64KB clustered PTEs. In contrast, with a conventional pagetable, sixteen times as many 4KB PTEs must be replicated. With the multiple pagetable approach, clustered pagetables require fewer tables. For example, one clustered table can be used for pagesizes 4KB to 64KB, and another for up to 1MB, and so on. With conventional pagetables, we will need as many tables as the number of pagesizes supported.

The clustered pagetable does not avoid the major complexities involved in supporting large pagesizes. Talluri et al. proposed using replicated clustered PTE to represent pagesizes larger than 64KB. Therefore, clustered pagetable implementation entails the complexity along the same lines as ours, namely looping over operations across subblock structures. Indeed it is true that a single lock could be used for each subblock of PTEs in the case of a clustered pagetable. With the conventional hashed table which is used in HP-UX [8], the locks have to be acquired individually for each base page PTE, and therefore, can entail high overhead. However, our implementation performs well despite this overhead.

We preferred using the replicated PTE method when implementing multiple pagesize support in the page directory (pagetable), as discussed in Section 4. Saving space was not compelling enough a reason for moving to clustered structures – we needed to gather performance data from an implementation to justify total redesign of the data structures. Regarding hashed pagetable without clustering, Talluri et al. raise several concerns. One concern is that the hash chains will be longer because a large page will use multiple base page entries. However, in our design we size the hash table based on the size of physical memory, and hence, the hash chains are small and no different from the case when only base page mappings are used. Another concern is that the entries corresponding to the subpages will be in different hash buckets thereby making operations that involve all the subpage entries less efficient. Our implementation does involve updating all the subpage translation entries in such cases as modifying a translation, and our system performs well despite this inefficiency.

2.5 Online Page Promotion

Another approach for reducing TLB misses employs online superpage (large page) promotion [18]. This work describes a technique for monitoring TLB miss traffic to decide when a superpage should be constructed. On each miss, page reference information (a set of counters) is updated to indicate the number of TLB misses that would not have occurred had the set of already assigned superpages been larger. Several online policies that perform suitable superpage promotions based on the counters, the TLB miss cost, and page copying cost are presented. The modifications to the TLB miss handler for the purpose of tracing does slow the miss handling. However, the overhead is absorbed into the significant performance gains made through page promotions.

While the online methods have advantages, the extent of modifications to the hardware independent layer may be significant. The key advantage of online methods is that superpages are created only when necessary, thereby ensuring that the working set size does not increase dramatically. In addition, the online approach appears to be less invasive, since the superpage awareness is confined to the hardware dependent layer. However, Romer et al. do not discuss all the details pertaining to page promotion. First, when their system is about to promote to a superpage whose subpages are not all resident, the nonresident pages must be brought in. Presumably, the fault-path could be repeatedly invoked to bring in the nonresident pages. Second, their system will need an allocator for allocating multiple pagesizes. Furthermore, demoting a superpage to subpages whenever the modified and referenced bits are queried from the hardware independent side, may not be desirable. For instance, paging out an unreferenced large page may be more efficient.

Despite the concern that OS support for multiple pagesizes would be invasive and complex, we concluded that by preserving the underlying 4KB data structures and using simple locking protocols, we had a promising approach that would lead to a successful implementation. We create large pages at fault-service time thereby eliminating the overhead from slower TLB miss handling as well as copying costs that the online promotion method entails. Since our approach will reduce the number of faults, it has the potential to offset some of the overhead from other large page related operations.

3 Data Structures for Large Pages

First we present an overview of PA-RISC and HP-UX VM architecture and data structures. Then we discuss the alternatives we considered with regard to representing large pagesizes and the approach we have taken.

3.1 VM Data Structures Overview

We begin with a description of addressing and access control in PA-RISC. Next we present the key data structures employed by the hardware dependent and the hardware independent components of the HP-UX VM subsystem. The hardware independent component is based on UNIX System V Release 2 and Release 3 [1].

The PA-RISC architecture defines a global virtual address space [14]. A global virtual address is made up of two components: a space identifier (space-ID), and an offset. The offset in turn is partitioned into a virtual page number, and a page offset. The PA-RISC 2.0 allows a 64 bit space-ID and a 64 bit offset, which are combined to generate up to a 96 bit global virtual address [10]. In the 32-bit and 64-bit HP-UX implementations, a process can access up to 4GB and 16TB respectively, using 4 space-IDs.

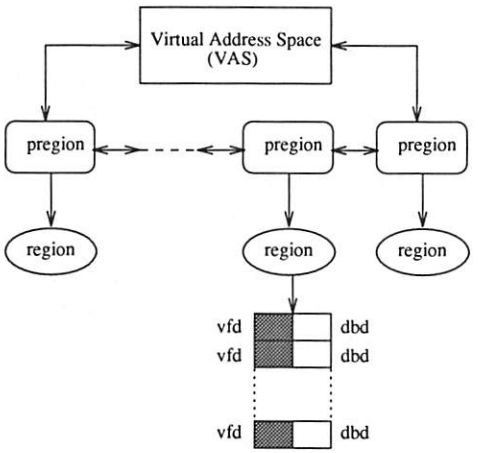


Figure 1: Hardware Independent VM – vas

Since the precision architecture uses global addressing, it employs a set of mechanisms to restrict what parts of this global space a process can access. The privilege level (0 through 3), access rights (read, write, execute), and a protection ID (PID) are used to control the access of a page by a process. The most privileged level of 0 is kernel mode, and the least privileged level of 3 is the user mode. Each process is assigned a set of PIDs, four of which are cached in control registers. To be allowed access to a page, the page's PID must match one of the process' PIDs.

Each process is associated with a *virtual address space* (*vas*) shown in Figure 1, which is made up of a list of *pregions*. Each pregion represents a range of virtual pages. As shown in Figure 1, each pregion points to a system-wide kernel data structure called a *region*. Each page in a region is associated with a *virtual frame descriptor* (*vfd*) that specifies the *page frame number* (*pfn*) and a *disk block descriptor* (*dbd*) that specifies the location of a page on disk. The vfd, dbd pair constitute the hardware independent pagetable entry. The vfds and the dbds are maintained in chunks of 32 pairs. The chunks are organized as a B-tree [5] for efficient access.

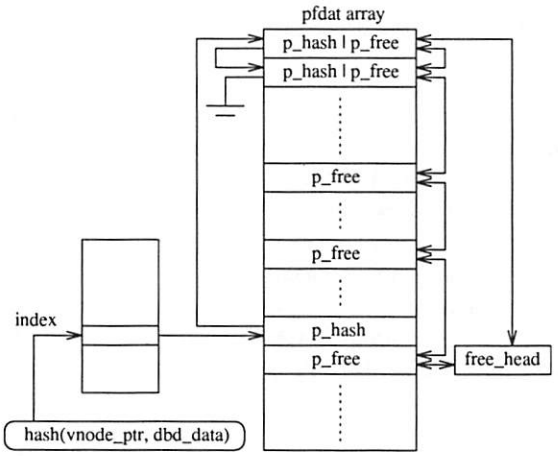


Figure 2: Hardware Independent VM – pfdat

Another central kernel data structure on the hardware-independent side called *pfdat* shown in Figure 2, is used to manage physical memory that can be allocated to processes on demand. Each physical page frame (pfn), which is 4KB in size is represented by a pfdat structure. The pfdat entries of physical pages that are available for subsequent allocation are placed on a doubly linked list referred to as the *freelist*. Page frames associated with space allocated on secondary storage are placed on the hashed *pagecache* list. These pages are caches of data on secondary storage. The pfdat structure does not hold a pointer to the region structure associated with the virtual page frame that is mapped to the pfn.

On the hardware-dependent side, a system-wide hashed page directory (pagetable) [8] referred to as the *pdir* is used to hold the virtual-to-physical address translation information. The pdir shown in Figure 3 contains one entry (*pde*) for every 4KB page of physical memory in the system, plus entries for mapping virtual I/O pages. The pdir layer performs operations

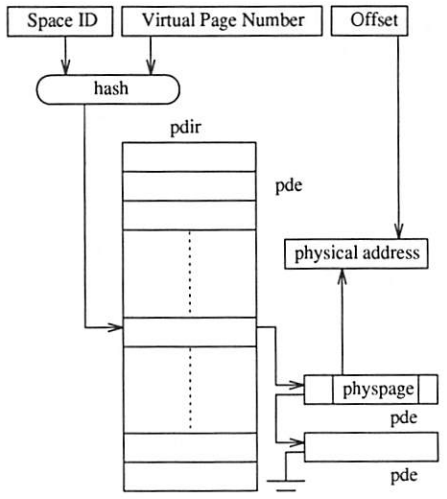


Figure 3: Hardware Dependent VM - pdir

to establish and manage the virtual-to-physical mappings for pages. Additionally, to manage address aliasing, it maintains a table of physical-to-virtual mappings. The hardware dependent layer (HDL) functions provide the interface for requesting such operations as adding and deleting translations.

The PA-RISC 2.0 architecture allows for either a separate or a combined data and instruction TLB. The PA-8x00 implementations use a combined TLB. Each TLB entry contains the virtual page number (tag), physical page number, an encoded 4-bit pagesize, access control information, and single-bit flags such as *dirty*, *break*, and *uncacheable*. A range of pagesizes from 4KB to 64MB (in multiples of four) are supported. If a TLB entry holds the matching virtual page number, based on the pagesize field, the corresponding 38 to 52-bit physical page number is concatenated with the 12 (4KB) to 26-bit (64MB) offset, to generate a 64-bit physical address. In the event of a TLB miss, a software miss handler fetches the translation from the pdir and inserts it into the TLB.

3.2 Representing Large Pagesizes

We considered two approaches for representing large pagesizes in HP-UX. One approach was to employ variable pagesize based structures, that is, to have each of the data structures represent variable pagesizes. Even with this method, we were not looking to redesign the VM system from scratch, and we wanted to reduce the extent of changes such as the pregion and the region algorithms. One implication of this requirement was that when using this method, each pregion/region represent a uniform pagesize. Another approach was to preserve the VM data structures based on 4KB pagesize, and represent a large pagesize as a collection of these base pagesize structures. Talluri et al. refer to this method as the replicate-PTE method [22] and the single-page-size framework [20] in the context of pagetables and the hardware independent VM module respectively. We will refer to it collectively as the replication method. After considering the benefits and the drawbacks of the two alternatives, we chose the replication approach.

While variable pagesized VM structures use less space and are updated efficiently, this method suffers from several drawbacks. First, changes are pervasive in that most of the VM system assumes systemwide single pagesize, and therefore, must be modified to use variable pagesized data structures. Second, since each pregion/region must represent a uniform pagesize, typically the number of pregions/regions that need to be traversed will increase. For example, when servicing a fault, if the large pagesize specified for a region cannot be allocated, the pregion/region must be suitably split so that a smaller pagesize can be used. The alternative of waiting for a large page to become available may lead to an unacceptably long page-fault latency. Also, since a large page must be aligned on the pagesize boundary, multiple pregions/regions may be needed for a virtual address range, to meet the alignment restrictions. Third, swap management will need to handle different sizes on swap space, and must minimize fragmentation, much like the physical memory allocator. Fourth, a large page protected by a single lock will lead to contention, when I/O operations are performed to non-overlapping portions of a memory region such as shared memory.

We chose the replication method for our implementation. This method is attractive for several reasons. Changes needed for multiple pagesize support are not pervasive – modifying attributes of part of a large page does not need splitting operations from pregion and region layers. Chances for contention are low, since many operations need to lock a specific 4KB pfdat even when the pfdat represents a member 4KB page of a large page. For example, examining a translation requires holding only one subpage pfdat lock. On the other hand, modifying a translation requires that locks on all subpage pfdat structures be held. Indeed, the replication method also suffers from some drawbacks. First, it does not take advantage of large pagesizes to reduce space used by the VM data structures. Second, locking, access, and update of data structures are inefficient – need to loop over the base pages of a large page. However, this looping overhead is no worse than when all the mapping are 4KB pagesize. We chose the replication method, because its benefits outweigh its shortcomings.

4 Hashed pdir Management

A large page is represented by multiple pde (page directory entry) structures, one each for a 4KB subpage. Each pde includes pagesize information. When a TLB miss occurs on a large page, the miss handler has no knowledge of the pagesize of the page on which the miss occurred. The miss handler locates the pde corresponding to the faulting address, and inserts it into the TLB. The replication method entails no additional complexity in the TLB miss handler, and the miss penalty remains unchanged in general, from the 4KB mapping case.

There is one caveat to the claim above that the TLB miss handler does not entail additional complexity – the handler for shared memory multiprocessors (MP) had to be modified to avoid disabling interrupts for too long. When a property of a translation for a large page is to be changed, the pdir management module must invalidate all the associated pdes, purge the TLBs, update the pdes, and then revalidate the pdes. If a TLB miss occurs on another processor in the meantime, it would be spin waiting in the TLB miss handler with interrupts disabled, until the subpage pde becomes valid again. To avoid holding off interrupts for too long, the MP TLB handler has been modified to enable interrupts, handle any pending interrupts, and then try accessing the pde again. This process is continued until the pde becomes valid.

Operations pertaining to a large page translation could involve access to a specific subpage or all the subpages. For example, the TLB miss handler will update information such as the *referenced* and the *modified* bits for the subpage pde on which the miss or the trap occurred. However, these bits indicate the status of the entire large page (if any) associated with this subpage. Therefore, the HDL functions must return the values accordingly, in response to queries from the hardware independent layer. To add a translation for a large page, pdes are allocated and added for each subpage. All operations that update a translation must update the pde for each subpage, and consequently, are more expensive. This overhead has not been a problem on the several benchmarks that we used for our performance measurements.

5 The Physical Memory Allocator

Support for multiple pagesizes places several new requirements on the physical memory allocator. First, the allocator must be able to allocate any of the pagesizes supported by the architecture. The page allocated must be aligned at a starting physical address that is a multiple of the pagesize. Second, the allocator must maintain the free and cached pages in such a way that fragmentation is minimized, and large pages can be found easily. The allocator must be efficient – its performance should not be much worse than the original 4KB page frame allocator.

We have implemented a binary buddy system allocator [13]. The allocator maintains the available memory pool as two subpools, the *uncached* subpool and the *cached* subpool. Pages in the cached subpool are linked to the pagecache list as discussed in Section 3.1. Each subpool has one *freelist* per pagesize. Only the first pfdat of a large page is linked to the appropriate freelist. The allocator maintains a pagesize field in the pfdat structure. Given a member pfdat of a large page, it is possible to find the first pfdat of the large page. The total count of pages allocated is maintained for each pagesize. Count of free pages in each of the cached and the uncached subpools is also maintained for each pagesize. These counters are updated as appropriate when pages are allocated, freed, or demoted. *Demotion* refers to breaking a large page mapping such as 64KB into smaller pagesize mappings such as 16KB. For instance, a process may request that the protection attributes of a memory range be modified, and this range may be a part of a large page. In this case, the VM subsystem must demote the large page, and then update the protection attributes for the demoted pages that lie in the requested range.

The buddy system reduces fragmentation, and increases the chance of finding a large page. In response to an allocation request, the allocator first searches the list for the pagesize requested, and if that list is empty, it searches the lists for bigger pagesizes. The uncached subpool is searched before the cached subpool, with the goal of preserving the cached pages as much as possible to facilitate reuse. In response to a request to free a page, the allocator attempts to locate the pfdat structure for the buddy page. If the buddy is on the same list, the allocator can coalesce.

The allocator employs different coalescing policies for the cached and the uncached subpools. The allocator coalesces the uncached pages as soon as they are freed. Coalescing can bubble up to the larger pagesize freelists. In contrast, the allocator uses a lazy approach to coalescing pages in the cached subpool. The freelist for each pagesize in the cached subpool is allowed to grow to a certain fraction of the total count allocated in that pagesize, before coalescing is performed. This fraction at this time is 25%. Experience with more workloads will be needed to determine suitable values for this watermark. The lazy approach reduces the amount of time spent in coalescing, given that the cached pages may be reused again.

The current implementation does not coalesce across the cached and uncached subpools. Once again, this choice was made to allow for reuse of cached pages.

Fragmentation is an issue despite the use of a buddy system allocator. Pervasive or transient heavy workloads can lead to fragmentation of the available memory pool. The allocator may not be able to find a large page, because one or more of the subpages may be in use. We are aware of some environments where memory fragmentation is not likely to occur, and others, where fragmentation could lead to low availability of large pages. Support for reducing fragmentation would involve paging out a certain 4KB page or copying it to a different page and freeing the source page, to create contiguous physical base pages.

6 Pagesize Hints

A pagesize hint is available to the fault-path from the region data structure. A region's pagesize hint is determined using one of two methods – neither of these methods require recompilation of the application. In the *chatr* method, a user specifies pagesizes for an executable's text and data regions to the *chatr* (change attributes) program. The chatr program places these pagesize hints in the executable header. When the executable is *exec*'ed, the region creation routine copies the hints from the executable header into the respective region structures. In the *transparent* method, the region creation routine computes the pagesize hint for a region based on the region size (number of 4KB pages), and saves it in the region structure. Hints computed using the transparent method are bounded by a minimum pagesize and a maximum pagesize, which are system tunables. The actual pagesize selected by the fault-path could be smaller than a region's pagesize hint, for reasons discussed in Section 7.1 Furthermore, if a large page allocation fails, the fault-path reverts to using a 4KB pagesize mapping.

Pregions that grow dynamically such as the heap, need additional support for exploiting benefits from large page mappings. Recall that in our implementation, we create large pages at fault-service time only – we do not perform online promotion of subpages into a large page. Since many existing applications tend to grow their heap in small increments such as 4KB, these pregions will be subsequently faulted in as small pages. To overcome this problem, we track pregions/regions that grow to a large size in small increments, and increase their growth rate so that large page mappings can be created at fault-service time. In this approach, a process that makes a break request of size such as 4KB through a *sbrk*() call could receive a larger break size such as 16KB. The scaled-up break value is determined by the prior history of break requests, and the data pagesize hint for the process. Subsequent requests from the process involving a break value that is smaller than what the kernel returned previously require no action from the kernel.

7 The Fault-path

The fault-path uses any of the hardware supported pagesizes when servicing validation or protection faults on various parts of a process address space including text, initialized data, uninitialized data, heap, stack, shared memory, and shared libraries. Faults on uninitialized data, heap, and stack segments are serviced using zerofilled memory. Faults on text and initialized data pages, if not found in the page-cache, require secondary storage access by the page-in handler of the associated filesystem. Faults resulting from copy-on-write or copy-on-reference sharing are resolved by copying the source page. The fault-path can create a large page mapping, when a process faults on a page that was previously paged out. In this case, the swap page-in handler may use a pagesize that is different from the one that was used prior to page-out.

In the next several sections, we describe our implementation, focusing on the representative aspects of multiple pagesize support. First, we present the infrastructure used in implementing zero-fill, filesystem page-in, and swap page-in. Then, we describe multiple pagesize support for filesystem page-in, followed by copy-on-write.

7.1 The Infrastructure

The fault-path uses three key interfaces. The pagesize selection interface determines the virtual pagesize to be used for servicing a fault. The vfd-fill interface allocates a large page, and fills the vfds with the corresponding pfns. The vfd-set interface updates status information in the vfds associated with a range of subpages.

The pagesize selection interface selects the pagesize that can be used, given a faulting 4KB page virtual address. It extracts the pagesize hint from the region structure, and lowers it if the available physical memory is less than 4 times the pagesize. Starting from this adjusted pagesize hint, this interface determines the pagesize that encompasses the faulting 4KB page. As outlined in Figure 4, a large page must meet alignment restrictions, and other conditions. The pagesize selected could therefore be smaller than the region's pagesize hint.

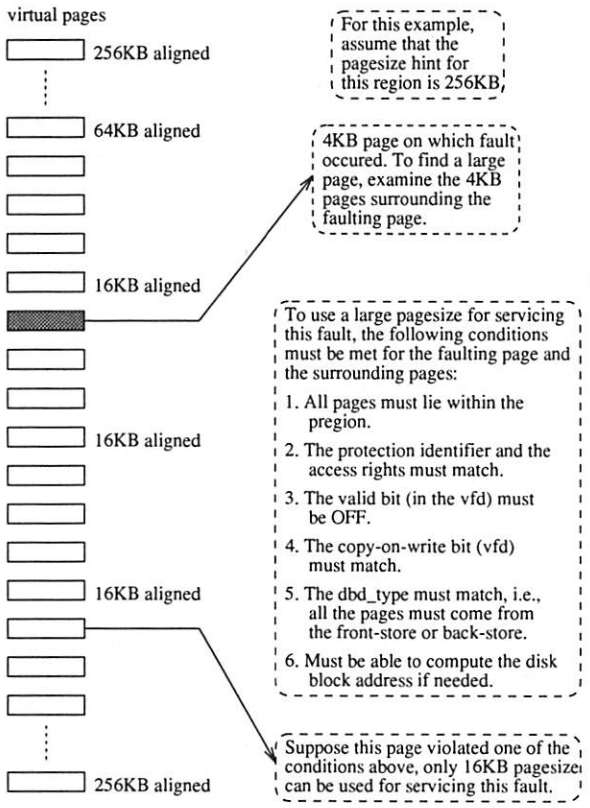


Figure 4: Finding a Large Virtual Page

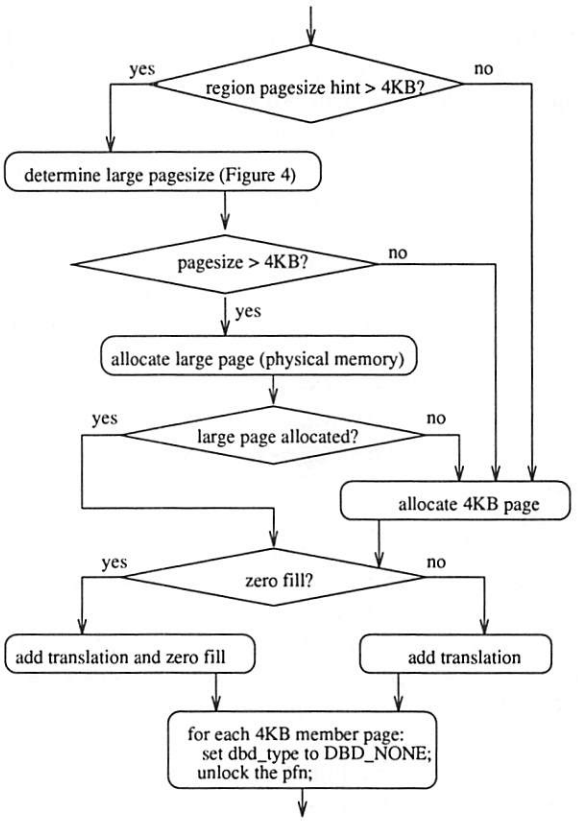


Figure 5: Creating a Zero-filled Large Page

The primary purpose of pagesize hint adjustment is to avoid depleting available memory by allocating large pages too aggressively. We would rather allocate a 1MB page and not a 4MB page, when 5MB of memory is available. This policy ensures that more medium-size large pages will get used in the system, instead of a few very large pages. Pagesize adjustment also increases the chance of a large page allocation request succeeding, because the allocator is more likely to find a 1MB page than a 4MB page, when 5MB of memory is available. It should be noted that since the number of pages available are low when in near memory-pressure conditions, that is, most of the memory is used by existing processes, the pagesize hint will likely get adjusted to 16K.

The vfd-fill interface makes an allocation request for a large page with the no-wait option, and if the allocation succeeds, fills the 4KB based virtual frame descriptors (vfd) with the corresponding 4KB page frame numbers (pfn). It should be noted that when a pagesize larger than 4KB is selected, the fault-path does not sleep and wait for that size to be allocated. Instead, it requests allocation with the no-wait option – the allocator will return failure if a page of that size is not readily available.

The vfd-set interface is used for setting or clearing the valid bit and the copy-on-write bit, in a range of vfds pertaining to a large page. Other looping operations involving large pages are implemented as macros, or as in-line code segments. The use of some of these interfaces is demonstrated in Figure 5.

7.2 Filesystem Page-in

Each filesystem has a page-in handler to service the fault pertaining to the respective filesystem's vnodes [12]. HP-UX 11.0 supports multiple pagesizes for UFS (UNIX File System), NFS (Network File System), and VxFS (Veritas journaling File System). Using the VFS (Virtual File System) VM initialization interface, all page-in handlers communicate whether or not they have been adapted for multiple I/O. Two of the significant tasks relevant to multiple pagesize support are finding a large page in the pagecache, and performing multiple I/O operations to bring in a large page.

The pagecache is examined after selecting the pagesize as discussed in Section 7.1. Prior to multiple pagesize support, it was necessary to look for only the faulting 4KB page in the pagecache. If the page is not found in the pagecache, then the page-in handler

must bring the page in from secondary storage. In the case of a large page, it is necessary to find all the subpages in the pagecache, to have a "pagecache hit". The 4KB pages that are found must be contiguous, and the first page must be aligned on the large pagesize boundary. If the large page found is bigger than the selected pagesize, it is demoted, and then used.

The UFS, NFS, and VxFS page-in handlers have been modified to expand their I/O, taking large pagesize into consideration. The original UFS and NFS page-in handlers make the assumption that the pagesize is smaller than the filesystem blocksize, and perform a single I/O. However, this assumption is no longer valid with large pagesizes. Support has been added to the UFS and NFS page-in handlers to perform multiple I/O operations to bring in a large page. The original VxFS page-in handler makes no assumptions about the relative sizes of pages and filesystem blocks; it already handles multiple I/O when bringing in clustered-4KB pages. Therefore, the only modifications to the VxFS page-in handler involve calls to VFS VM interfaces, for the purpose outlined in the next paragraph.

It is preferable to hide the notion of large pages within the VFS VM routines. To meet this goal, existing VFS VM interfaces have been modified, and also new interfaces have been added. The page-in handlers themselves deal only with I/O expansion; they are oblivious to the pagesize being used. They make calls to the VFS VM interfaces, which determine if the I/O expansion meets the pagesize-related restrictions. If the restrictions are not met, the VFS VM routines return a result to indicate that the page-in handler must retry for a clustered-4KB page-in.

7.3 Copy-on-write

The original implementation of the module that performs copy-on-write operations, handled one 4KB page only. Here the source (physical) page is located, and the fault is resolved with or without copying the page. In the no-copy case, the source page itself is used to resolve the fault. Extending the no-copy case to handle large pages is straightforward, requiring only looping of relevant operations over the 4KB subpages.

The complexity arises in the copy case because of certain assumptions that were made in the original implementation. Operations such as allocating the physical memory for the destination page, and allocating kernel virtual space to map the source page for copying, are always expected to succeed. Some of the copy-on-write related data structure and translation manipulations are interspersed with these resource allocation operations. But in the case of a large page, the allocation request with no-wait option could fail. In addition, allocating a large page from the kernel virtual space could also fail. Either of these failures would necessitate that certain operations be backed out, which is not easily done if we simply extend the original implementation to handle large pages.

To handle the copy case for large pages, the copy-on-write module was modified to perform all resource allocations upfront. With this implementation, in the event of any resource allocation failures, the source page is demoted to 4KB pages, and copy-on-write is performed on the faulting 4KB page only. If large page resource allocations do succeed, other operations pertaining to the copy case are carried out by looping over 4KB subpages.

8 Page Replacement

The pageout daemon process uses a two-hand clock algorithm [15, 24] and a *not-recently-used* policy for page replacement. The algorithm uses two clock hands walking memory, with the first hand (*age-hand*) clearing the referenced bits of pages, and after a certain delay, the second hand (*steal-hand*) sampling the referenced bits. HP-UX bases its scanning on pregions/regions rather than physical frames, for reasons that include locking, and the need for the ability to avoid paging out from high priority processes. Also, UNIX System V Release 2 [1], on which the HP-UX VM system is based, takes this approach. As the age-hand scans pregions in the list, it ages 1/16 part of each pregion at a time.

Both the age and the steal hands must recognize large page boundaries. If the daemon encounters a large page, it completes the scan or pageout of the entire large page. The pageout daemon does not need to look for and free pages of specific sizes, because the fault-path never sleeps waiting for specific pagesizes to be allocated. This design choice is in tune with our goal of avoiding unnecessary complexity unless there is demonstrable performance gain.

In choosing a candidate for pageout, "fairness" regarding large pages and small pages is a concern. A large page has a better chance of being in referenced state, when compared to a relatively smaller page. In only one case, the pageout clearing-scan breaks a large page into smaller pages. This demotion is performed, if a portion of this large page has been wired down via a memory locking interface such as *mlock*(2). Otherwise, the daemon will scan and push pages at whatever pagesize they are. This solution involves the risk that larger pages may be kept around at the expense of pushing smaller pages, especially 4KB pages to disk.

Whether this situation is frequent and problematic will become clear from the feedback from users of real world applications. The alternate solution of breaking a referenced large page during clearing-scan into pages of the next smaller size may not be the best choice. Without support for page promotion, the benefits of a large page would be lost, leading to performance degradation from TLB misses, once memory pressure eases.

9 Performance Evaluation

We measured the performance of the benchmarks and one commercial application (Verilog-XL from Cadence Design Systems, Inc.) described in Table 1. The benchmarks and Verilog-XL were chosen, because their performance improves significantly when using large pagesize mappings. We focus on the reduction in TLB miss overhead due to large pagesizes. We do not discuss other effects such as reduced validation and protection faults, and large I/O. While these additional factors may be interesting and worthy of analysis, they are outside the scope of this paper.

All the benchmarks were run on a HP 9000 Series 800 machine with a 180 MHz PA-8000 processor. The Verilog-XL application was run on an earlier version of the machine with a 160 MHz PA-8000 processor. The processor includes 96 combined data and instruction TLB entries. The results that we report here are under no-paging conditions, that is, the working sets always fit within available physical memory. We made the performance measurements on the 32-bit version of the HP-UX 11.0 operating system. Table 2 describes the performance metrics.

We used the chatr program discussed in Section 6, to set the data and text pagesize hints in an executable header. While it is possible to specify distinct pagesize hints, we used identical hints for both data and text. Once an executable header has the hints set, the kernel will not attempt the transparent pagesize-hint selection for the process. To prevent large pages from being used for other processes that may not have been chatr'ed, we set the minimum and maximum pagesize kernel-tunables to 4KB, thereby disabling the transparent pagesize-hint selection altogether. This setup ensures that performance benefits from TLB miss reduction come from using large pages in our benchmarks alone.

The impact of TLB misses, and the benefits from using large pagesizes for the SPEC95 benchmarks apsi, compress, and vortex are shown in Table 3. For the benchmark apsi, the predominant component of the TLB misses is due to initialized data references. With

Benchmarks	Description
apsi	determines temperature, distribution of pollutants; part of the CFP95 group from the SPEC95 suite [19];
compress	compresses and uncompresses data in memory; part of the CINT95 group from the SPEC95 suite;
vortex	database program; part of the CINT95 group from the SPEC95 suite;
VMbench	3 benchmarks Nastran, ProE, and Verilog simulate VM behavior of commercial applications NASTRAN (mechanical analysis, Computerized Structural Analysis & Research Corporation), Pro/ENGINEER (mechanical design, Parametric Technology Corporation), and Verilog-XL (electronic simulation, Cadence Design Systems, Inc.);
Verilog-XL	commercial engineering application (electronic simulation) from Cadence Design Systems, Inc;

Table 1: Applications

Metrics	Description
TLB Misses (1000's)	Number of TLB misses in 1000's. Data collected using Hewlett Packard's *cyclemeter* tool.
TLB Time (m:s)	Estimated TLB miss overhead in minutes:seconds. Calculated using the average miss handling overhead of 70 cycles for a PA-8000.
Total Time (m:s)	Benchmark's total execution time in minutes:seconds. Measured using the *time* command.
User Time (m:s)	Time spent in user mode by the benchmark. Measured using the *time* command.
Mem Usage (4KB)	Total physical memory in 4KB pages mapped to the benchmark's address space, as determined at the end of the run.
Pagesize Distribution	Count of each of the pagesizes mapped to the benchmark's address space, as determined at the end of the run.

Table 2: Performance Metrics

16KB pagesize, TLB misses are reduced significantly. Compared to 4KB pagesize, the memory usage increased only by 5%. Beyond the 16KB chatr pagesize, large pages do get allocated, and the memory usage increases considerably. However, there is no further

apsi					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	132747	0:52	2:47	2:46	600
16KB	392	***	2:05	2:05	630
64KB	51	***	2:05	2:05	649
256KB	37	***	2:05	2:05	713
1MB	36	***	2:05	2:05	905
4MB	35	***	2:05	2:05	905

compress					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	53378	0:21	2:24	2:22	8947
16KB	115	***	2:03	2:03	8957
64KB	48	***	2:03	2:02	8985
256KB	33	***	2:03	2:02	9129
1MB	29	***	2:04	2:03	9257
4MB	28	***	2:03	2:02	9769

vortex					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	156828	1:01	3:25	3:24	14571
16KB	81192	0:32	2:57	2:56	15083
64KB	41401	0:16	2:41	2:40	15427
256KB	14668	0:06	2:30	2:29	15603
1MB	242	***	2:23	2:22	15795
4MB	50	***	2:23	2:22	16563

Table 3: Results for apsi, compress, and vortex

VMbench Nastran					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	739179	4:38	9:56	9:53	11188
16KB	299314	1:56	6:40	6:37	12142
64KB	83808	0:33	4:56	4:54	13378
256KB	969	***	4:18	4:17	13730
1MB	112	***	4:18	4:16	13922
4MB	101	***	4:18	4:16	14434

VMbench ProE					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	230495	1:30	3:06	3:05	20557
16KB	129450	0:50	2:16	2:15	21543
64KB	65804	0:26	1:46	1:45	22563
256KB	16628	0:06	1:21	1:20	22627
1MB	95	***	1:13	1:12	22819
4MB	31	***	1:13	1:12	23587

VMbench Verilog					
chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB
4KB	408487	2:39	5:22	5:16	53127
16KB	178919	1:10	3:42	3:39	53802
64KB	46225	0:18	2:43	2:41	53810
256KB	1407	***	2:21	2:19	53858
1MB	105	***	2:20	2:19	54050
4MB	88	***	2:20	2:19	54306

Table 4: Results for Nastran, ProE, and Verilog

performance improvement, because TLB misses are no longer a significant overhead. For the benchmark compress, with 4KB pagesize, the predominant component of TLB misses are due to dynamically-allocated (dynamic) data references. TLB misses are reduced with 16KB chatr pagesize. The increase in memory usage is negligible. With 16KB and larger chatr pagesizes, TLB miss overhead becomes insignificant. The benchmark vortex also benefits from dynamic data being mapped using large pagesizes. The TLB miss overhead is reduced gradually as the chatr pagesize is increased from 4KB to 1MB. The reduction in total time closely follows the reduction in the estimated TLB miss overhead.

As shown in Table 4, all three VMbench benchmarks benefit from using large pagesize mappings for dynamic data. All three benchmarks are trace driven, and a command-line parameter specifies the granularity of dynamic allocation (*malloc* size). The malloc size of 16MB is used in the Nastran benchmark, and hence large pagesizes can be exploited for the heap. The benchmark ProE on the other hand, makes malloc requests in small varying increments, and yet, benefits from large pages because of heap growth-rate adjustment discussed in Section 6. The benchmark Verilog uses a malloc size of 16KB. Nonetheless, it benefits from larger chatr pagesizes, also because of heap growth-rate adjustment.

Results from Cadence Design's Verilog-XL, a real world application, are presented in Table 5. Verilog-XL is a digital simulator that allows an engineer to test the logic of a design. Verilog-XL suffers from TLB misses primarily due to dynamic data references. The text and the initialized data sizes constitute less than 5% of the total memory usage.

Some interesting observations can be made from

chatr Pagesize	TLB Misses 1000's	TLB Time m:s	Total Time m:s	User Time m:s	Mem Usage 4KB	Pagesize Distribution						
						4KB	16KB	64KB	256KB	1MB	4MB	16MB
4KB	601116	4:23	38:32	38:19	33818	33818						
16KB	317135	2:19	35:59	35:47	33924	1124	8200					
64KB	104463	0:46	33:33	33:21	33996	1044	14	2056				
256KB	23840	0:10	32:36	32:25	34193	1113	14	8	514			
1MB	6403	0:03	32:27	32:16	34362	1114	16	10	12	126		
4MB	3523	0:02	32:23	32:12	34641	1113	14	8	13	3	31	
16MB	2844	0:01	32:20	32:10	36666	1114	16	10	12	3	1	8

Table 5: Performance Results for Verilog-XL

Table 5. First, as the chatr pagesize is increased from 4KB to 256KB, the performance gain realized is higher than the gain from TLB miss reduction. One possible reason for the additional gain could be reduced number of virtual faults. Second, Verilog-XL does not show as dramatic a performance improvement as VMbench Verilog, the synthetic version. This lack of correlation is due to VMbench Verilog simulating only the memory reference behavior, and not the computation performed by the real application. Third, both Verilog-XL and VMbench Verilog realize most of the performance gain when 256KB pagesize is used. Fourth, even with large chatr pagesizes, over 1000 4KB mappings remain. Some of these mappings belong to shared libraries that were not chatr'ed to use large pagesizes.

Table 5 demonstrates the benefits of heap growth-rate adjustment in a real application. This technique facilitates the use of large pagesizes, in spite of the application making requests to grow the heap by small increments. Note that with heap growth-rate adjustment, the wasted heap allocation can be at most the chatr pagesize for program data. The 256KB pagesize offers most of the performance improvement for Verilog-XL. This pagesize improves performance by over 15% with only a 1% increase in memory usage.

10 Summary and Conclusion

We have implemented multiple pagesize support in HP-UX, using the existing 4KB pagesize based hardware dependent and hardware independent VM data structures. By using this "replication" based representation, we are not taking advantage of large pagesize based structures that are more efficient in space, and possibly more efficient in time. On the other hand, our approach entails no more overhead than when all mappings are 4KB pagesize; our primary goal was to reduce the TLB miss overhead. We use a buddy system allocator for allocating and freeing multiple pagesizes. Our implementation creates large pages at fault-service time for zero-filled memory, page-ins from secondary storage for UFS, NFS, and VxFS, page-ins from swap space, and copy-on-write. We have a page replacement module that handles multiple pagesizes.

Our performance measurements show that despite the seemingly high overhead of operating on 4KB based data structures, our approach to multiple pagesize support offers significant performance improvement for a variety of benchmarks and a real world application.

Acknowledgments

Thanks to Orran Krieger, the shepherd for our paper, for his detailed and insightful comments that helped improve the presentation. We are grateful to the anonymous referees, Anurag Acharya, and Joe Barrera for their comments on the paper. We appreciate the feedback on an early version of the paper, from Eric Hamilton, Duncan Missimer, and Hal Prince. Thanks to Venkatesh Radhakrishnan for his comments on a near-final version. Jun Su helped us with measuring the performance of Verilog-XL.

Bill Taylor was the project manager. Eeman Wong wrote the functional and reliability tests. Jyothy Reddy contributed to the initial version of the pageout path to handle multiple pagesizes. Cliff Mather implemented kernel support for pagesize hints, changes to HDL functions, and added hooks for collecting large page statistics. Kurt Peterson performed design reviews and code reviews for the project, and implemented the final version of the pageout path. Balakrishna Raghunath implemented pdir support, and the allocator. Indira Subramanian investigated alternatives for multiple pagesize support, implemented the fault-path, and wrote this paper.

References

[1] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., 1986.

[2] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 243–253, November 1994.

[3] P. Bannon and J. Keller. Internal Architecture of Alpha 21164 Microprocessor. In *Compcon Digest of Papers*, pages 79–87, March 1995.

[4] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 114–123, May 1992.

[5] D. Comer. The Ubiquitous Btree. *ACM Computing Surveys*, pages 121–137, June 1979.

[6] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, pages 323–333, May 1968.

[7] D. Greenley et al. UltraSPARC: The Next Generation Superscalar 64-bit SPARC. In *Compcon Digest of Papers*, pages 442–451, March 1995.

[8] J. Huck and J. Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 39–50, May 1993.

[9] T. Kagimasa, K. Takahashi, and T. Mori. Adaptive Storage Management for Very Large Virtual/Real Storage Systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, pages 372–379, May 1991.

[10] G. Kane. *PA-RISC 2.0 Architecture.* Prentice-Hall, Inc., 1996.

[11] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. Virtual Memory Support for Multiple Page Sizes. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS)*, pages 104–109, October 1993.

[12] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–247, June 1986.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, third edition, 1997.

[14] R. B. Lee. Precision Architecture. *IEEE Computer*, pages 78–91, January 1989.

[15] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[16] *MIPS R10000 Microprocessor User's Manual, Version 2.0.* MIPS Technologies, Inc., 1996.

[17] J. C. Mogul. Big Memories on Desktop. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS)*, pages 110–115, October 1993.

[18] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 176–187, June 1995.

[19] SPEC. SPEC Newsletter, September 1995.

[20] M. Talluri. Use of Superpages and Subblocking in the Address Translation Hierarchy. Ph.D. Thesis, University of Wisconsin-Madison Computer Sciences, August 1995.

[21] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 171–182, October 1994.

[22] M. Talluri, M. D. Hill, and Y. A. Khalidi. A New Page Table for 64-bit Address Spaces. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 184–200, December 1995.

[23] M. Talluri, S. Kong, M. D. Hill, and D. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 415–424, May 1992.

[24] U. Vahalia. *UNIX Internals – The New Frontiers.* Prentice-Hall, Inc., 1996.

SimICS/sun4m: A VIRTUAL WORKSTATION

Peter S. Magnusson[1], Fredrik Dahlgren[2], Håkan Grahn[3], Magnus Karlsson[2], Fredrik Larsson[1], Fredrik Lundholm[2], Andreas Moestedt[1], Jim Nilsson[2], Per Stenström[2], Bengt Werner[1]

[1]{psm,fla,am,werner}@sics.se

Swedish Institute of Computer Science
Box 1263, SE-164 28 Kista
SWEDEN

[2]{dahlgren,karlsson,dol,j,pers}@ce.chalmers.se

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg
SWEDEN

[3]Hakan.Grahn@ide.hk-r.se

Department of Computer Science
University of Karlskrona/Ronneby
SE-372 25 Ronneby
SWEDEN

Abstract

System level simulators allow computer architects and system software designers to recreate an accurate and complete replica of the program behavior of a target system, regardless of the availability, existence, or instrumentation support of such a system. Applications include evaluation of architectural design alternatives as well as software engineering tasks such as traditional debugging and performance tuning.

We present an implementation of a simulator acting as a virtual workstation fully compatible with the sun4m architecture from Sun Microsystems. Built using the system-level SPARC V8 simulator SimICS, SimICS/sun4m models one or more SPARC V8 processors, supports user-developed modules for data cache and instruction cache simulation and execution profiling of all code, and provides a symbolic and performance debugging environment for operating systems.

SimICS/sun4m can boot unmodified operating systems, including Linux 2.0.30 and Solaris 2.6, directly from snapshots of disk partitions. To support essentially arbitrary code, we implemented binary-compatible simulators for several devices, including SCSI, console, interrupt, timers, EEPROM, and Ethernet. The Ethernet simulation hooks into the host and allows the virtual workstation to appear on the local network with full services available (NFS, NIS, rsh, etc). Ethernet and console traffic can be recorded for future playback.

The performance of SimICS/sun4m is sufficient to run realistic workloads, such as the database benchmark TPC-D, scaling factor 1/100, or an interactive network application such as Mozilla. The slowdown in relation to native hardware is in the range of 25 to 75 (measured using SPECint95). We also demonstrate some applications, including modeling an 8-processor sun4m version (which does not exist), modeling future memory hierarchies, and debugging an operating system.

1. Introduction

A target computer system typically runs a mixture of operating system and application code, which interact in a complex, fine-grained manner to solve the tasks at hand. This interaction between operating system and application, and between operating system and the underlying hardware, today constitutes a difficult domain in computer science and engineering. The overall performance of a system is largely determined by this interaction, and it is required to function correctly to provide a stable platform.

To explore this interaction in any detail is difficult, especially since designers are frequently interested in not just understanding an existing system, but to explore alternatives. A fully simulation-based approach has the advantage of essentially being able to model any architecture and gather any statistic. Since we are concerned with the software level in general, and in the interaction between software and key hardware resources in particular, the principal approach is that of instruction set simulation.

Running operating system code on simulators has long been a common practice in the computer manufacturing industry. Little of this work has been done in academia, and consequently general tools of this character have not been available to a broader community. Also, the techniques used have, to our knowledge, often been inefficient, with a focus on hardware development rather than making available a practical tool for operating system designers. Thus they would at most support small benchmarks, require expensive custom hardware, or require impractical running times or resources for usage. For similar reasons, the tools have had little, if any, support for performance tuning (such as profiling).

Foremost among published work is g88 (Bedichek 1990), which combined threaded code (Bell 1973, Klint 1981) with simulation of simplified devices. g88 could

boot and run the Unix kernel using device drivers for these simplified device models. g88 could support debugging but not performance tuning, and had only limited support for computer architecture work.

To address these limitations we have designed a simulation platform on top of which it is possible to execute and analyze unmodified complex application and operating system software with a decent performance. This paper describes the simulation platform and demonstrates its efficiency by analyzing the performance of a database application that is run on a 4-CPU multiprocessor simulator that models the sun4m architecture.

The contributions of this paper are twofold. First, we implement a full system model built on the SimICS simulator, allowing the advanced debugging and profiling features of SimICS to be applicable to operating system work. Second, we implement a system level simulator that runs completely unmodified operating system binaries, actually booting from dumps of the partitions that would boot a target machine. As far as we are aware, this has never been described in the open literature.

The rest of this paper is organized as follows. In Section 2, we present instruction set simulation as a general technique. In Section 3 we present SimICS, our simulator kernel, which constitutes a flexible platform on which a complete target system simulator can be built. The sun4m architecture is such a target, and in Section 4 we describe the principal components, that together with SimICS forms a full simulator. We describe a few example uses of the simulator in Section 5 and discuss its performance in Section 6. We discuss previous and related work in Section 7, with a particular emphasis on SimOS, a system level simulator with similar capabilities and goals as SimICS. We conclude in Section 8.

2. System level instruction set simulation

Instruction set simulators run a program by simulating the effects of each instruction on a target machine, one instruction at a time. Instruction set simulators are attractive for their flexibility: they can, in principle, model any computer, gather any statistic, and run any program that the target architecture would run, including the operating system. They easily serve as backends to traditional debuggers as well as architecture design tools such as cache simulators.

For their flexibility, instruction set simulators have long been popular in computer architecture research. There they help designers understand the tradeoffs involved in architectural decisions by simulating the effects on user programs.

Naturally, this flexibility comes at a cost—instruction set simulators are often slow, easily over 3 orders of magnitude slower than native execution. Such poor performance severely hampers their practicality, limiting them to toy benchmarks or very patient users. This has prompted several efforts to improve the performance of traditional simulation or to find alternate methods. This work has met with some success: several fast instruction set simulators have been developed over the last several years (Bedichek 1990 and 1995, Veenstra 1994, Cmelik and Keppel 1993, Witchel and Rosenblum 1996).

Besides the issue of performance, a full implementation is also complicated by the difficulty of recreating the execution environment. To run a given program, either we can emulate the underlying operating system faithfully, or we can bypass this difficulty entirely by running the operating system directly.

Unfortunately, the execution environment of modern systems is large. Running the operating system as an "application" is the obvious alternative, but is challenging since this requires faithful emulation of the system-level architecture. Earlier work along these lines therefore replaced complex devices with pseudo devices—devices with simple behavior (Bedichek 1990, Magnusson 1993a, Rosenblum *et al* 1995, Werner *et al* 1997).

There are several problems with not implementing proper device simulators. Firstly, they are frequently significant to overall system performance. Especially in the light of ever improving microprocessor speeds, I/O performance is important and becoming more so every day. Since our purpose is to improve the overall performance of systems, we cannot exclude these devices.

Secondly, an important use of this class of tools is to support the development and tuning of hardware dependent components of the operating system, and this of course requires an accurate emulation.

Finally, from a practical standpoint of distributing and supporting a simulator, reliance on pseudo devices adds the complication of needing to distribute a modified operating system. This may require source code access and/or special licenses for the user, which can be difficult for the research community. In addition, it is a task that needs to be repeated for each operating system that is intended to run on the simulator.

3. SimICS

SimICS is an instruction-set simulator developed at the Swedish Institute of Computer Science (SICS). It simulates one or more SPARC V8 processors, and supports multiple physical address spaces, system-level code, and emulation of the SunOS 5.x ABI for direct analysis of user-level programs. The performance of SimICS is acceptable even for large problems, with a slowdown of around 25-75 per simulated processor. SimICS itself is sequential, allowing it to be fully deterministic, a crucial feature for an instrument.

SimICS allows a program to be studied interactively, both for debugging and for profiling. Of primary interest, SimICS can profile data and instruction cache misses, translation look-aside buffer misses, and instruction counts. These figures can be weighted, sorted, and related to source code lines, allowing the programmer to quickly zoom in on the portions of code that consume resources.

SimICS has evolved over 7 years, and has absorbed almost 20 man-years of effort.

3.1. SimICS interpreter

The core of SimICS is a variation of threaded code (Bell 1973). Interpreters execute programs by running a central fetch-decode-execute loop. Some simple design ideas improve performance. Firstly, the target program, in object code format, is translated to an intermediate format, which is in turn interpreted. Whereas the target instruction set is designed for interpretation by hardware, the intermediate format is designed to be easy for software. During execution, this intermediate code is then cached. A variety of data structures keeps track of when to regenerate intermediate code.

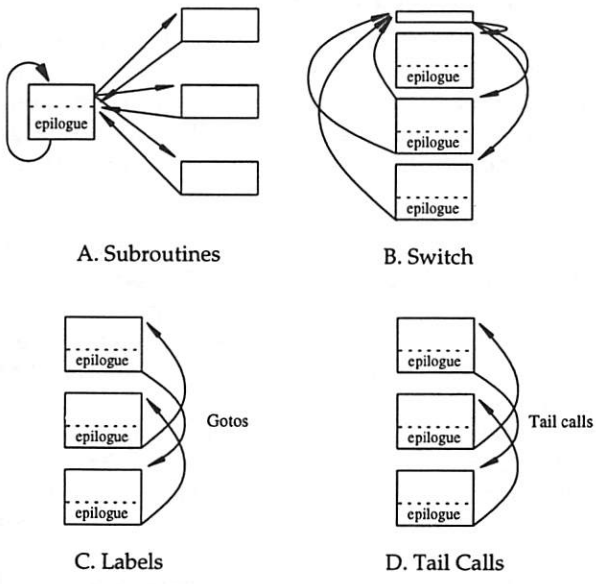


Figure 1 – Interpreter models

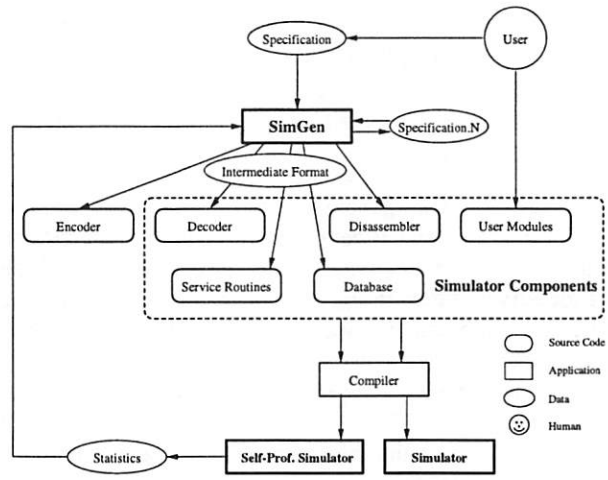


Figure 2 - SIMGEN Overview

For each intermediate format instruction there is a small segment of code, called a *service routine*, that emulates the effects of that instruction, as well as performing any administrative tasks for the simulation, such as event queues, instruction pipeline, etc.

There are several ways of dispatching these service routines; Figure 1 shows the four most common. These are: subroutines called from an inner loop, a large "switch" statement, directly addressable labels, and function calls relying on tail call optimization. SimICS primarily uses addressable labels using GCC (Stallman 1992), but can also run using tail recursion.

The process of implementing, and supporting, an industrial-grade complete instruction set simulator is a significant task. An instruction set will typically require several hundred different service routines. The core interpreter of SimICS is therefore implemented using a metatool, SIMGEN, which automates a range of tasks related to interpreter design (Larsson *et al* 1997). SIMGEN works from a simulation-oriented specification of the target instruction set, see Figure 2. Currently, it will design the intermediate format, and then generate a decoder, disassembler, encoder, and set of service routines. Metatools such as SIMGEN have been in use for some time, the contribution of SIMGEN is that it can generate faster interpreters than is practical to do manually. One way it accomplishes this is by generating versions of the interpreter that gathers service routine usage statistics, which it can then take as input to regenerate a faster interpreter.

SIMGEN essentially solves two porting issues. Firstly, the support of new instruction sets is greatly simplified, since SIMGEN works from a high-level specification. An earlier, handwritten SPARC V8 interpreter (Samuelsson 1994) consisted of some 10,000 lines of C macros, and was reimplemented using only 2,000 lines of specification. SIMGEN has also been used to generate

interpreters for different versions of the APZ212, a proprietary embedded CISC processor (Egeland 1995). The result is now a component in Ericsson's test environment, a product used by several thousand software developers.

Secondly, it can generate different interpreter cores from the same specification. For example, SIMGEN can generate interpreter cores corresponding to any of the alternatives shown in Figure 1. The performance for the various alternatives varies with processor/compiler combination, as well as varying over time. Also, different compilers support different combinations. For example, GCC 2.7 does not support tail call elimination, and directly addressable labels cannot be expressed in ISO C.

Most service routines are simple, typically 10-30 host processor instructions. This sets an upper limit on performance for this technique of about 20 times slower than native execution. To obtain significantly better performance, optimized runtime code generation techniques can be used, and several prototype versions of SimICS have supported them (Magnusson 1993b, Christensson 1997). In practice, such techniques are not yet superior to the interpreter design in SimICS. A full discussion of this interesting, but parenthetical, issue is beyond the scope of this paper.

3.2. Accuracy vs. efficiency

SimICS is primarily a functional simulator, meaning that it does not model timing at a detailed level such as CPU pipelines. A simulator such as SimICS can, however, complement a cycle-accurate model by providing subtraces consisting of a sequence of hardware events that derive from coarse elements in the target system (caches, CPUs, TLBs, etc). In other work, we've demonstrated the efficacy of this division of labor (Werner and Magnusson 1997). It requires that a simulator such as SimICS efficiently keep track of *non-volatile* state, i.e. processor state that changes slowly, such as cache contents. The objective of SimICS itself is thus primarily (a) to model the target system sufficiently accurately to run any software and (b) to efficiently model non-volatile state with high granularity.

3.3. Memory simulation

A crucial component of a system level simulator is the simulation of memory. Memory operations are difficult to handle, since not only are they both frequent and complex, but in a simulator we also wish to gather additional information. A significant portion of the design effort and complexity in SimICS lies in how it simulates memory (Magnusson and Werner 1997).

Briefly, SimICS collapses a range of operations into a common (optimistic) path using a Simulator Translation Cache (STC). Figure 3 illustrates the principle. The service routine for the memory operation will perform an inlined lookup in the STC. If it "hits" in this data structure, it can proceed directly. The inlined lookup is carefully designed and consists of nine host (SPARC) instructions. This implicitly or explicitly includes a physical to logical translation, a TLB lookup, a protection check, a watchpoint check, an alignment check, a data cache lookup, location of the physical storage in the simulator, and a profile count of the target address.

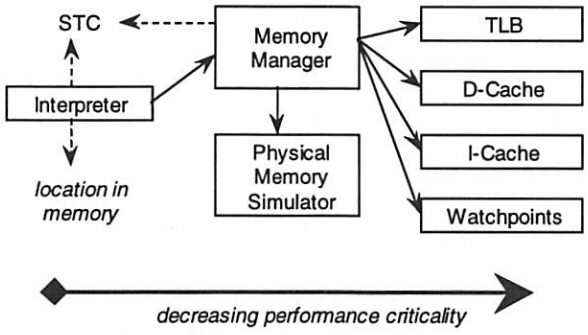


Figure 3 - Memory Simulation

The instruction cache is supported in a similar manner, but is expanded to handle a from-to jump cache, so as to support not only cache modeling, but accurate profiling, a variety of breakpoint types, and, in future, support for various branch prediction models. Because of it's similarity to the STC, this design is called the I-STC, and is described further in (Magnusson 1997).

3.4. SimICS as a Platform

Figure 4 shows a schematic overview of SimICS as a platform. SimICS itself consists of an interpreter core which can be extended in a variety of directions using published programming interfaces. We take advantage of dynamically loadable modules to run-time extend SimICS in this manner.

These extensions are primarily of two types. First, devices can be added to build a full system. Such device modules load themselves as generic device objects, and the user can instantiate them at the command line (SimICS has a simple command line interface). Devices are memory-mapped, i.e. once instantiated SimICS will redirect any memory accesses to the requested physical memory region to that device. Devices include Ethernet, to communicate with a real network, a console, for interactive serial terminal sessions, and disk subsystems. The principal devices needed for the sun4m architecture that we implemented

are described in Section 4. The second important extension type is memory hierarchy modules, allowing the user to add new cache models.

For software engineering tasks, SimICS can run as a backend to a symbolic debugger. We use a modified version of GDB 4.16 as our preferred interface. In this mode, SimICS will fake stack contents, manage multiple address spaces, etc.

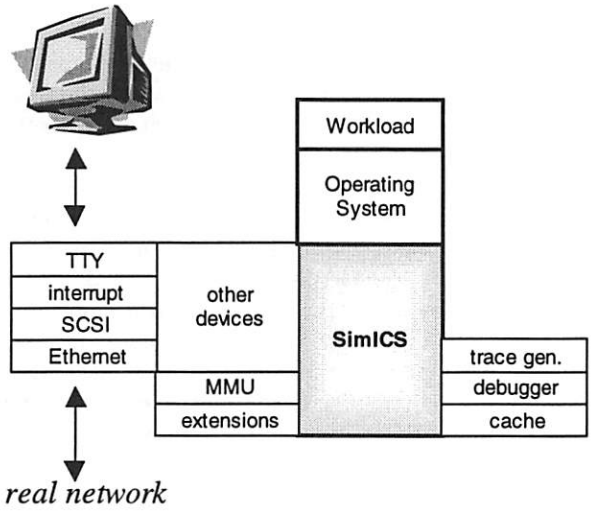


Figure 4 - SimICS Platform

3.5. Profiling support in SimICS

The essential benefit of running within a simulator is that the complete state is available for inspection. The design of SimICS has largely focused on techniques for gathering detailed information on the execution. Probably the most useful feature is *profilers.*

A profiler gathers and presents statistics that are related to a (physical) memory address range. A simple example is an *execution profiler*, which counts how many times an instruction at a particular address has been executed.

Profiler values are shown whenever the user lists source code. The profilers currently supported by SimICS include those listed in Figure 5. Profilers support building various generic analysis tools on top of them. For example, the **`prof-weight`** command produces the output in Figure 6. Each profiler has an associated weight, such as **`$SIM_TLB_MISS_WEIGHT`**. The user can interactively change weight values. This allows the user to explore a very large set of data and focus on one issue at a time. For example, in Figure 6, the TLB miss weight has been set to 10 ("Column 4").

(a) instruction cache misses
(b) write cache misses (data)
(c) read cache misses (data)
(d) translation look-aside buffer misses
(e) branches to the instruction
(f) branches from the instruction
(g) count of instruction execution
(h) flag for instruction execution
(i) reads from memory address
(j) writes to memory address

Figure 5 - SimICS profilers

The **`prof-weight`** command thus calculates a linear sum over the entire memory, sorting and displaying the largest values. In the figure, we have asked SimICS to calculate the weights in address intervals of 32 bytes and to display details for the top 20.

The example is from an analysis of the infamous SPECint92 benchmark *eqntott*. By using SimICS in the role of a traditional profiling tool, we improved the performance of *eqntott* by a magnitude. The profile information can also be related to code, as we will show in Section 5.3.

```
(gdb-simics) prof-info
Active profilers, from 'left to right':
Column 1:  Instruction cache misses caused by program line
           ($SIM_INSTR_MISS_WEIGHT = 10.000000)
Column 2:  Cache misses (writes) caused by program line
           ($SIM_WRITE_MISS_WEIGHT = 1.000000)
Column 3:  Cache misses (reads) caused by program line
           ($SIM_READ_MISS_WEIGHT = 8.000000)
Column 4:  TLB misses passed on to Unix emulation
           ($SIM_TLB_MISS_WEIGHT = 10.000000)
Column 5:  Number of (taken) branches *to* the code block
           ($SIM_TO_WEIGHT = 0.000000)
Column 6:  Number of (taken) branches *from* the code block
           ($SIM_FROM_WEIGHT = 1.000000)
Column 7:  Count of instruction execution (based on branch arcs)
           ($SIM_PC_WEIGHT = 1.000000)
Column 8:  Number of addresses from which instr have been fetched
           ($SIM_INSTR_WEIGHT = 0.000000)

(gdb-simics) prof-weight 32 20
Weighted profiling results:
  Physical    Virtual     ( source )
  0x00005c20  0x00011c20 (pid 1001) 518199272.00
  0x00005c40  0x00011c40 (pid 1001) 366859495.00
  0x00005c60  0x00011c60 (pid 1001) 335490415.00
  0x00005c00  0x00011c00 (pid 1001)  38342452.00
  0x00005d20  0x00011d20 (pid 1001)  33332216.00
  0x000084a0  0x000144a0 (pid 1001)  21651844.00
  0x00005d40  0x00011d40 (pid 1001)  20545152.00
  0x00005c80  0x00011c80 (pid 1001)   9771702.00
  0x000084c0  0x000144c0 (pid 1001)   7240831.00
  0x00005be0  0x00011be0 (pid 1001)   5890173.00
  0x00006460  0x00012460 (pid 1001)   5768754.00
  0x00005ca0  0x00011ca0 (pid 1001)   4945636.00
  0x00008480  0x00014480 (pid 1001)   4405064.00
  0x000084e0  0x000144e0 (pid 1001)   4155135.00
  0x000064e0  0x000124e0 (pid 1001)   4059607.00
  0x00017b20  0x00023b20 (pid 1001)   3921297.00
  0x00008900  0x00014900 (pid 1001)   3569070.00
  0x00005ba0  0x00011ba0 (pid 1001)   3353840.00
  0x00008c60  0x00014c60 (pid 1001)   3244719.00
  0x00008500  0x00014500 (pid 1001)   3215813.00
Sum:            1397962487.00 (90%)
Not shown:       160930057.00 (10%)
System total:   1558892544.00
```

Figure 6 - `prof-weight` listing

```
proc = sym.val("practive")
  while proc != 0:
    pr_str = "((proc_t *)0x%x)" % proc
    pid = sym.val("%s->p_pidp->pid_id" % pr_str)
    cmd = sym.str("%s->p_user->u_comm" % pr_str)
    ctx = sym.val("((srmmu_t *)%s->p_as->a_hat
          ->hat_data[0])->s_ctx" % pr_str)
    print "pid = %3d (ctx = %3d) %s" % (pid, ctx, cmd)
    proc = sym.val("%s->p_next" % pr_str)

pid = 234 (ctx =  20)  "mozilla"
pid = 233 (ctx =  19)  "mozilla"
pid = 231 (ctx =   3)  "csh"
pid = 227 (ctx =  22)  "ttymon"
pid = 225 (ctx =  16)  "sh"
pid = 224 (ctx =   2)  "sac"
pid = 199 (ctx =  13)  "utmpd"
pid = 189 (ctx =  18)  "sendmail"
pid = 184 (ctx =  14)  "nscd"
pid = 178 (ctx =  15)  "cron"
pid = 165 (ctx =   4)  "syslogd"
pid = 161 (ctx =  12)  "automountd"
pid = 146 (ctx =   9)  "lockd"
pid = 144 (ctx =   7)  "statd"
pid = 139 (ctx =  10)  "inetd"
pid = 110 (ctx =  11)  "ypbind"
pid =  99 (ctx =   8)  "keyserv"
pid =  97 (ctx =   5)  "rpcbind"
pid =  26 (ctx =   6)  "dhcpagent"
pid =   3 (ctx =   0)  "fsflush"
pid =   2 (ctx =   0)  "pageout"
pid =   1 (ctx =   1)  "init"
pid =   0 (ctx =   0)  "sched"
```

Figure 7 - Scripting example

3.6. Scripting interface

Work is currently in progress with extending SimICS to support various scripting languages. We use SWIG (Simplified Wrapper Interface Generator) to generate the glue code between the script interpreter and SimICS, making it easy for us to experiment with more than one language. Python and Tcl are the ones that SimICS currently support. With a script language, users can write their own commands, for example to traverse data structures in the source of a program being run. Figure 7 shows an example of a Python function that lists all processes in Solaris, printing their pid, MMU-context and name. In the example, **sym** is a module that handles symbolic information. The two functions **val** and **str** in this module return the value and the string, respectively, for a specified symbol.

3.7. Validation

As a tool, SimICS has a variety of uses. The methods of validation differ with application. For example, for coarse grain characterization of a workload, internal cross-checks occur in SimICS that can be consulted by the end user. These give a reasonable assurance that aspects such as instruction count is accurate. Each cache model used offers a new validation problem. If the cache model corresponds to an existing system, we can validate by comparing with hardware traces, as has been done with such applications (Werner and Magnusson 1997). If the cache model is of a future design, then SimICS does not offer a silver bullet. The principal method employed is to compare a specialized cache model with a generic (parameterized) model, leveraging off the fact that SimICS, being completely deterministic, can exactly duplicate the workload.

4. Modeling the sun4m system

In this section, we describe the most significant components of the simulated sun4m architecture, namely disk (SCSI) and network (Ethernet), as well as a mechanism for recording and playing back I/O traffic.

We emphasize that all devices were developed to the point of fidelity where both of our target operating systems, Linux 2.0.30 and Solaris 2.6, would boot and run completely unmodified.

Currently, we short-circuit the first phase of the boot process, namely the boot PROM, which we have reverse-engineered and written from scratch, rather than dump PROM contents to a binary as we did in earlier work (Magnusson 1993a). There were two principal reasons for this: we wish to be able to distribute the whole environment, and the PROM is copyrighted; and we are not interested in the error checking and initialization aspect of device fidelity that is exercised by the PROM. Our fake PROM works in a simulator-"aware" fashion, with support for parsing a target architecture description, thus simplifying handling of target variations.

SimICS exports an interface for adding new devices. Devices are loaded and instantiated dynamically. There is also a variety of facilities for debugging either a device driver, a device simulator, or both. This includes separate, dynamic history buffers for all devices, where the device simulator can log a descriptor of the effects of the command.

4.1. SCSI and connected disks

We modelled the FAS100 chip prescribed by the sun4m architecture. SCSI is relatively tricky to model properly since it is highly asynchronous. Disk transactions are handled in multiple steps, and several different tasks can be outstanding at any time. As a consequence, the SCSI simulator is by far the most complex device, requiring 4,500 lines of C.

Disk contents are modeled by taking dumps of real partitions. These dumps are naturally rather large, and are treated as read-only input to the simulator. Changes resulting from SCSI writes are kept internally in the simulator in a delta structure, which can be read or written to a file. This allows multiple simulator sessions to use the same set of original dump files. It also simplifies configuration management: to set up a particular session, you boot the operating system on the simulator, log in, do system administration or install software using 'ftp' or similar, shut down the simulated operating system in an orderly manner, and then save the delta file. This new delta file can then subsequently be used for new runs.

4.2. Network Support

The sun4m model in SimICS supports connectivity to a real network on the Ethernet level. The model contains a simulated Ethernet device mapped into the memory space of the simulated machine. The simulated device is given the same Ethernet address as the real interface on the host machine. To separate packets destined for the target and the host OS, the IP address is checked. When started, the code for the Ethernet device spawns two processes running as root. One is used for reading and one for writing raw Ethernet frames through the network interface on the host machine. The read process filters out all packets for the simulated machine, i.e. IP packets and ARP requests containing the simulated IP address. The host OS also receives these packets, but throws them away since the IP address does not match. The write process simply sends frames to the network. We use the Packet Capture library (libpcap) from LBNL to send and receive Ethernet frames from within user processes. Figure 8 shows the architecture of the network connection.

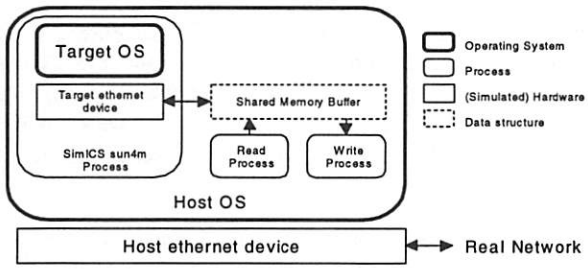


Figure 8 - Network Support Overview

We can also use DHCP, which allows us to create boot disk snapshots that are independent of the local network, and thus transportable across domains.

We have run a variety of network applications on the simulated machine with success: rlogin, ftp, automount, X11, etc. By using X11 we can run applications with graphical interfaces, without having to simulate a display device. Also, ftp and nfs are two simple ways to get program files into the simulated environment.

4.3. Handling Asynchronous Input

All asynchronous input for the simulated machine can be saved for later playback. This currently includes tty (keyboard) and network data. By saving and replaying input, the same simulation session can be repeated with identical behavior.

All external sources of input are polled by SimICS and the data is sent to a special device, called the recorder. The recorder works in either recording or playback mode. In recording mode, the recorder saves the data to a file together with a timestamp before passing it on to the appropriate device. In playback mode the polling functionality is disabled, instead the recorder reads data from a file, and then passes it on as normal.

Using this recording facility, sessions with asynchronous events can easily be debugged. First, they are run with live input that is recorded. Then, in playback mode, code can be single-stepped and the simulation can be interrupted for inspection of state without disturbing the simulated session.

5. Examples of Usage

SimICS/sun4m is currently being used to support a range of activities. In this section, we present some example uses.

5.1. Simulating with Large Workloads

To demonstrate the capability of SimICS/sun4m in handling large working loads, we made a few simple measurements on booting and running Solaris 2.6. In the first example we ran for 26 billion simulated instructions, which took roughly 3 hours of simulation time on a 250 MHz UltraSparc.

In Table 1 we see the result from this first run, an interactive session consisting of four phases. First, the operating system booted on a single processor system. Next, we run the graphics program 'xv', doing a simple image manipulation. In the third phase, the system is idling, with only system processes running, and last we run 'ftp' reading several megabytes of data, followed by a Netscape Communicator session. The numbers give a coarse characterization of what is happening in the machine. We also note that the simulated *mips* figure falls with an increased load, as exceptions and memory writes become more frequent, events that are expensive both on the target machine and the simulator.

In the second run, shown in Table 2, we booted the Solaris 2.6 operating system on a 4-processor sun4m system. As we can see from the number of writes as well as the exception count, the work appears well-balanced over all processors. CPU 0 does some more memory operations, which can be expected since it is the processor running the initial boot sequence. We reach a multi-user login prompt after 1,03 billion instructions, compared to 1,40 billion in the single processor case, indicating some parallel work during the boot process.

Phase	Time (sec)	Instructions	mips	Instr/exception	Instr/read	Instr/write
Boot	2,100	1,403,150,579	0.67	566	5.36	12.4
xv	1,560	5,708,931,035	3.66	6,547	4.12	140.1
Idle	420	2,057,365,964	4.89	13,048	4.03	261.2
Ftp, NS	7,080	16,583,003,191	2.34	2,856	4.14	68.5

Table 1 - Large Unipro Workload

	CPU 0	CPU 1	CPU 2	CPU 3
Reads	221,240,785	200,868,331	202,759,327	203,967,084
Writes	88,717,788	59,773,968	64,189,093	63,220,767
Exceptions	25,201	30,774	34,610	30,693
Instructions	1,028,765,700	1,028,765,700	1,028,765,700	1,028,765,700

Table 2 - Parallel Boot of Solaris 2.6

5.2. Multiprocessor Architecture Studies

One important application of the SimICS/sun4m platform is to use it for evaluating design alternatives for multiprocessors. As a case study, we have evaluated the memory hierarchy of a shared-memory multiprocessor running a database application. The multiprocessor has four nodes, each containing a processor with a two-level cache hierarchy, a memory module, and a network interface. The memory modules of all nodes constitute the global, shared physical memory space, so that any processor can access any memory module.

On a cache miss, a block of memory is requested from the memory module where it is allocated. The ideal case would be if all data accessed by the processor is allocated in the memory module within the same node, instead of having to be sent over the network. While this can be the case for code and private data (such as the stack), it is near impossible for data shared between multiple nodes. One technique to reduce the amount of requests to memory in other nodes would be to add a remote cache (Zhang and Torrelas 1997), i.e. an additional level of cache entirely used to cache data from other nodes.

When using the SimICS/sun4m platform to evaluate the effectiveness of adding remote caches as a complement to the ordinary two-level cache hierarchy, a memory system simulator of the target system is developed as a separate module using a predefined interface to SimICS. When SimICS is run, the memory system simulator is loaded, and the memory references will now become visible to it. In addition, the memory system simulator will also be able to control the execution of each processor, so that a correct processor stall time will be modeled according to the latencies of the memory system.

Using this methodology, we have evaluated a four-node multiprocessor with and without remote caches. The simulated multiprocessor executes the same operating system and database handler binaries as we are running on our real workstations. In the experiments, we used Linux 2.0.30, which supports up to four processors, and the PostgreSQL v.6.1 client-server database handler from Berkeley (Stonebraker *et al* 1990).

The application of the database handler was query #6 of TPC-D with the scaling factor of 1/100 (20 MB of database tables). The multiprocessor was used as a database server, executing the same query on all processors individually in parallel. The database data is originally on disk, and the pages are allocated when accessed so that they will be distributed among the nodes. The sizes of the L1 and L2 caches were 16 KB (direct-mapped) and 512 KB (4-way set-associative), respectively.

We measured the total number of requests over the network for different sizes of the remote cache. The number of network messages was reduced by only 1.0% for a remote cache of 1 MB, and 1.2% for a remote cache of 2 MB. Experiments using scientific benchmark applications have indicated a much larger gain from using remote caches, so we decided to analyze the effects in more detail. The amount of data accessed by each node is 17 MB. By analyzing the temporal locality of the data, i.e. how much the data is re-used after it has been accessed by the node for the first time, we discovered that 88% of the memory blocks are not reused after they have been replaced from the L1 cache. We traced these references back to their sources and found that 95% of them originated from only 16 instructions in *memcpy()* in the kernel. Most of them were used by a sequential scan operation in the database application.

The sun4m architecture is only defined for up to four processors, which is an assumption explicitly used in the source code of Linux. In order to do architectural evaluations of systems with more than four processors using the Linux OS, we used the SimICS/sun4m platform to extend both Linux and the sun4m architecture to be able to support up to 16 processors. In

terms of Linux, it could be efficiently debugged using the SimICS platform. As a result, we were able to execute the same database system binaries (PostgreSQL running TPC-D) on a 16-node multiprocessor, and evaluate effects of novel memory system designs for the new architecture as well as the effects on the modified operating system such as lock contention.

As the above examples show, the SimICS/sun4m platform is capable of evaluating different design alternatives as well as explore novel architectural features and organizations for existing operating systems and applications. Moreover, it is also an efficient platform for modifying programs as well as operating systems and evaluate the effects of such modifications. Since the platform can execute any binary available for the Solaris 2.6 or Linux 2.x for the sun4m architecture, we are also able to evaluate commercial applications for which we do not have the source code. However, when the source code is available as in the above example, we are able to trace effects in the hardware/software interaction back to the source code of the application program as well as system software.

5.3. Mozilla on Solaris

To demonstrate the combination of user and system mode debugging, we have begun evaluating Mozilla running on Solaris, all on top of SimICS. We use Mozilla 5.0b1, a binary that when statically linked with debug information is over 60 MB. The support for symbolic debugging in SimICS handles multiple memory contexts, allowing the user to debug several programs, including the operating system, at the same time. In the following example we did measurements on Mozilla, starting at a push on the reload button and ending when the page (http://www.sics.se) was fetched from a real server and fully rendered in a window on another machine.

```
Phys page Instr.   Of total
0x2043000 53833494 0.251324
0x0e89000 53052056 0.247676
0x3188000 13675682 0.0638455
0x2046000 12303672 0.0574402
0x288f000  7453213 0.0347956
0x2001000  7047036 0.0328994
0x158c000  6494650 0.0303205
0x2034000  4365093 0.0203786
0x2112000  4327131 0.0202014
0x2006000  3245785 0.0151531
...
```

Figure 9 - Top page usage by Mozilla

The reload needed a total of 214 million SPARC instructions to complete. Figure 9 shows a list of pages with the highest count of executed instructions, and the fraction of the total number. As the list shows, 50% of all instructions executed can be found on only two pages, the rest are spread over 1059 other pages. A reverse memory translation (SimICS command **srmmu-reverse**) finds the first page in context 0, mapped by the kernel, and the second page in context 19, mapped by Mozilla. Zooming in on these pages reveals the *idle()* function in Solaris and the function *il_quantize_fs_dither()* doing Floyd-Steinberg dithering in Mozilla. Figure 10 shows some lines from the source, with profiling information, for the latter function. The profiling values shown are, from left to right: (a) I-cache misses, (b) D-cache write misses, (c) D-cache read misses, (d) branches to the block, (e) branches from the block, (f) count of instructions executed, and (g) a count of assembler instructions in the block. The cache statistics reflect a 16D/20I first-level cache configuration, 4-way and 5-way respectively with 32-byte cache lines. Note that this is only an example. To find out where Mozilla actually spends most of the time for a task, the command **prof-weight** should be used, taking into account the time for misses in caches and the TLB. Also, the binary should be compiled with more optimization that in this example.

```
      (a) (b)  (c)    (d)     (e)      (f)  (g)
244   179  0    0     282       0      564   2        dir = 1;
245                                                   /* => entry before first column */
246   272  0    0       0       0     1974   7        r_errorptr = cquantize->fserrors[0] + x_offset;
247     0  0    0       0       0     1974   7        g_errorptr = cquantize->fserrors[1] + x_offset;
248    70  0    0       0       0     1974   7        b_errorptr = cquantize->fserrors[2] + x_offset;
249                                                 }
250
251                                                 /* Preset error values: no error propagated to first pixel ...
252    54  0    0     281       0     1689   3    r_cur = g_cur = b_cur = 0;
253
254                                                 /* and no error propagated to row below yet */
255     0 13    0       0       0     1689   3    r_belowerr = g_belowerr = b_belowerr = 0;
256     0  0    0       0       0     1689   3    r_bpreverr = g_bpreverr = b_bpreverr = 0;
257
258   325  0    0  270261  270803  1085464   8    for (col = width; col > 0; col--) {
...
...
267     1  0 3900  270283       0  2432160   9        r_cur = RIGHT_SHIFT(r_cur + r_errorptr[dir] + 8, 4);
268    43  0 7987      60       0  2432160   9        g_cur = RIGHT_SHIFT(g_cur + g_errorptr[dir] + 8, 4);
269     1  0 2841      55       0  2432160   9        b_cur = RIGHT_SHIFT(b_cur + b_errorptr[dir] + 8, 4);
```

Figure 10 - Lines from the il_quantize_fs_dither() function with profiling data

	caches	*go*	*m88ksim*	*gcc*	*li*	*ijpeg*	*perl*	*vortex*
Native execution (sec)	N/A	3.2	0.5	8.1	1.0	8.3	14.5	13.0
Native MIPS		160	260	150	190	240	160	190
Sim 1 (sec)	Infinite	84.5	19.2	267.7	33.0	216.8	574.5	491.8
MIPS		6.0	6.9	4.6	5.6	9.2	4.1	4.9
Slowdown		x26	x38	x33	x32	x26	x40	x38
Sim 2 (sec)	16k/20k	123.9	23.9	545.9	52.9	257.6	810.1	980.8
MIPS		4.1	5.5	2.3	3.5	7.7	2.9	2.5
Slowdown		x39	x48	x67	x52	x31	x56	x75

Table 3 - SimICS Performance

6. Performance of SimICS/sun4m

The performance of a simulator is critical to its practicality. Measuring system level performance is difficult since it requires duplicating the workload. Let us first begin with an uncomplicated performance measurement, namely running in *user mode* on the simulator and comparing the execution time of SPECint95 programs on target and host, using the *train* data set. In user mode, SimICS emulates SunOS 5.x system calls using a compatibility layer.

Table 3 shows the resulting relative performance of SimICS over native execution. The timings were performed on an Ultra Enterprise, with the median of five *time* measurements shown (we've omitted *compress* since its native execution time was too small). The table shows a range of 26-75 in performance for two configurations of SimICS.

The first configuration, Sim 1, is with infinite data and instruction caches. In the second configuration we simulate a small, on-chip cache with 16kbyte 4-way set associative data cache and a separate, 20kbyte 5-way set associative instruction cache, corresponding to the SuperSPARC processor's on-chip caches, and a 64-entry unified TLB. All simulation runs generated full profiling (listed in Figure 5).

As the caches get smaller, the frequency of expensive events increases causes our slowdown to deteriorate. The baseline performance with minimum activity is close to the expected peak performance of the interpreter technique that we use, approximately 20. The performance loss for more realistic resource restrictions remains reasonable, within a factor of three of peak.

The *mips* numbers in Table 3 can be compared with the numbers given earlier in Table 1. We see that the ranges in raw interpreter performance is similar, leading us to conclude that the slowdown range of 26-75 is also representative for the full sun4m simulation environment.

7. Previous and Related Work

System level simulation for performance modeling is a longstanding tradition in industry. See, for example, (Canon *et al* 1979) for some early work. The first corresponding work in academia that we know of is the implementation of g88, which partly originated in industry. It was subsequently placed in the public domain and the design details published (Bedichek 1990). g88 modeled a uniprocessor M88100-based system with a mixture of real and pseudo devices, and could boot an operating system (Unix). gsim, the predecessor to SimICS, extended g88 to include support for multiple processors with shared physical memory (Magnusson 1992 and 1993a).

SimICS is a rewrite of gsim, primarily to model the SPARC architecture, but also to implement a faster, more portable interpreter core, as well as provide a more structured environment for system level simulation research in general.

A more recent tool, SimOS, models a MIPS-based multiprocessor (Rosenblum *et al* 1995 and 1997, Witchel *et al* 1996). SimOS can boot and run Irix. Newer versions of SimOS model other processors, such as the Alpha (Barroso *et al* 1998).

Both SimOS and SimICS have pursued similar goals and have thus, inevitably, arrived at similar solutions on many issues. For instance, both tools allow adding end-user memory hierarchy models, support copy-on-write disk images, can run off a local network as a virtual workstation, and provide tools and hooks for non-intrusively studying the behavior of the workload.

However, SimOS and SimICS have emphasized different aspects of simulation in pursuing performance. Both SimICS and SimOS are designed with hybrid techniques in mind, i.e. the intention is to model different sections of an execution at different levels of accuracy, thus gaining in performance but, using various sampling techniques, losing only marginally in accuracy. SimOS might thus have three CPU simulators, where the first focuses on quickly booting the OS, the second on warming the caches, and the third

on modeling processor pipelines, etc (Rosenblum et al 1995). The middle stage is needed since warming caches requires long traces. SimICS assumes that this middle stage will be the principal bottleneck in using the tool, and has thus focused on fast modeling of cache hierarchies (Magnusson *et al* 1995), which can complement a more detailed processor model (Werner *et al* 1997).

The performance goal of SimICS is to be fast when gathering detailed information on common hardware events. Today this includes a full profile of TLB, data cache, and instruction cache misses as well as instruction execution count, all at the granularity of single instructions. This information gathering is all subsumed in the 31-75 slowdown range (median 52) given earlier ("Sim 2"). The closest level of granularity and speed combination reported for SimOS would indicate a slowdown of around 130 (Herrod 1998a) for similar work.[1] This SimOS performance figure is for a large (1 Mb) second level cache, which stresses the simulator significantly less than the 16K/20K D/I caches used for the SimICS measurement. This level is more comparable to the "Sim 1" level above, with a slowdown of 26-40. In addition, SimOS does not maintain an execution profile. So despite providing both more detail and under a higher pressure of event frequency, SimICS is approximately 3 times faster than SimOS.

The SimOS instrumentation, on the other hand, is implemented in a more general manner. For example, SimOS supports general annotations allowing arbitrary script routines to be triggered by a variety of hardware events. This is a powerful tool for exploring program behavior. It allows the user to introduce problem-specific semantics for classifying hardware events. Thus, the performance advantage of SimICS over SimOS is largely an effect of a specialized approach beating a general approach. We are in the process of adding similar functionality to SimICS, and believe that this will not hamper the current specialized tools.

[1] Note that this performance is different from what is reported in (Witchel and Rosenblum 1996). The SimOS group found it worth trading off some of Embra's speed for maintainability, ease of debugging, and improved functionality (Herrod 1998b). This estimate is thus based on current SimOS behavior, namely a slowdown of 38-49 for "rough characterization mode" compounded by an overhead of 284% for classifying the hardware events.

8. Conclusions

We have presented a system level simulation environment that mimics the sun4m architecture from Sun Microsystems with sufficient fidelity to run full operating system workloads directly from disk partition dumps, including Linux 2.0.30 and Solaris 2.6. The environment furthermore supports symbolic debugging, performance tuning, and memory hierarchy evaluation tasks. The overall performance is better than two magnitudes slower than native execution, which is sufficient to run realistic benchmarks, including SPECint95 and TPC-D.

We expect SimICS/sun4m, and future environments like it, to play a significant role in both computer architecture and operating system design work.

9. Acknowledgements

Magnus Christensson got Linux 2.0.30 to boot on SimICS at SICS, and also ported SimICS to Linux. Björn Grönvall graciously gave of his time to explain a variety of Unix esoterics. This work has been supported by the SICS Framework Programme, Sun Microsystems, and Ericsson Infotech.

10. Availability

SimICS/sun4m is available for the research community at "http://www.sics.se/simics". The current distribution includes Linux boot disk images. This package is also included on the conference CD-ROM.

Bibliography

Anderson, J. M., L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. 1997. Conitinuous Profiling: Where Have All the Cycles Gone? Technical Report, 1997-016a, Digital Systems Research Center, September.

Barroso, L. A. and K. Gharachorloo. 1998. System Design Considerations for a Commercial Application Environment *First Workshop on Computer Architecture Evaluation Using Commercial Workloads.* In conjunction with the Fourth International Symposium on High Performance Computer Architecture (HPCA-4), Las Vegas, Sunday Feb. 1, 1998.

Bedichek, R. C. 1990. Some efficient architecture simulation techniques. In *Proceedings of Winter '90 USENIX Conference*, pp 53-63.

———. 1995. Talisman: Fast and accurate multicomputer simulator. In *Proceedings of the '95 ACM SIGMETRICS Conference,* pp 14-24.

Bell, J. R. 1973. Threaded Code. *Communication of the ACM*, 16(6):370-372.

Canon, M. D., D. H. Fritz, J. H. Howard, T. D. Howell, M. E. Mitoma, and J. Rodriquez-Rosell. 1979. A Virtual Machine

Emulator for Performance Evaluation. *Seventh Symposium on Operating System Principles,* Pacific Grove, California, Dec 10-12. As reprinted in *Communications of the ACM* 23, no. 2 (Feb.): 71-80.

Christensson, M. 1997. Techniques for runtime code generation in instrumented instruction set simulators. Masters Thesis, Royal Institute of Technology, Department of Teleinformatics.

Cmelik, R. F. and D. Keppel. 1993. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06.

Egeland, T. 1995. APZ 212 20 – The New High-end Processor for AXE 10. *Ericsson Review.* 72(1):5-12.

Herrod, S. A. 1998a. Using Complete Machine Simulation to Understand Computer System Behavior. February. PhD Thesis, Department of Computer Science, Stanford University.

———. 1998b. Personal communication, April 2nd, 1998.

Klint, P. 1981. Interpretation techniques. *Software - Practice and Experience*, 11(9):963-973.

Larsson, F., P. S. Magnusson, and B. Werner. 1997. SimGen: Development of Efficient Instruction Set Simulators. SICS Research Report R97:03, November.

Magnusson, P. S. 1992. Efficient simulation of parallel hardware. Masters thesis. Royal Institute of Technology (KTH), Stockholm, Sweden.

———. 1993a. A design for efficient simulation of a multiprocessor. In *Proceedings of MASCOTS*, pp 69-78.

———. 1993b. Partial Translation. SICS Technical Report T93:05.

———. 1997. Efficient Instruction Cache Simulation and Execution Profiling with a Threaded-Code Interpreter. In *Proceedings of the Winter Simulation Conference (WSC97).*

Magnusson, P. S. and B. Werner. 1995. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, pp 62-73.

Rosenblum, M., S. Herrod, E. Witchell, and A. Gupta. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, pp 34-43.

Rosenblum, M. S., E. Bugnion, S. Devine, and S. Herrod. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM TOMACS Special Issue on Computer Simulation.*

Samuelsson, D. 1994. System Level Interpretation of the SPARC V8 Instruction Set Architecture, SICS Research Report R94:23.

Stallman, R. M. 1992. Using and Porting GNU CC, version 2.0 (15 February 1992). Free Software Foundation, Mass., USA.

Stonebraker, M., L.A. Rowe, and M. Hirohama. 1990. "The implementation of POSTGRES," in IEEE Transactions on Knowledge and Data Engineering, March, vol.2, (no.1):125-142.

Veenstra, J. E. and R. J. Fowler. 1994 MINT: A front end for efficient simulation of shared memory multiprocessors. In *Proceedings of MASCOTS* '94, 201-207. January.

Werner, B. and P. S. Magnusson. 1997. A hybrid simulation approach enabling performance characterization of large software systems. In *Proceedings of MASCOTS 97*, pp 73-80.

Witchel, E. and M. Rosenblum. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the '96 SIGMETRICS Conference*, 68-79. ACM Press.

Zhang, Z. and J. Torrellas. 1997. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA, in Proceedings of the 3rd International Symposium on High-Performance Computer Architecture, pp 272-281, February.

High-Performance Caching With The Lava Hit-Server

Jochen Liedtke Vsevolod Panteleenko Trent Jaeger Nayeem Islam

Thomas J. Watson Research Center
IBM
Hawthorne, NY 10532
{jochen,vvp,jaegert,nayeem}@us.ibm.com

Abstract

With the development of new client-server computing models, such as thin clients and network computers, the performance of servers becomes a bottleneck. In these models, servers support a large number of clients. They download significant amounts of data to their clients in the form of graphics, executables (e.g., applets), and video. We present an architecture for building high-performance server systems that can efficiently serve large local clusters of NCs or other clients. The key component in our architecture is a generic cache module that is designed to fully utilize available bus bandwidth. Our experiments show that such a server system can achieve throughput rates of up to 36,000 transactions per second. We detail the design and implementation of the generic cache component, describe its use in the implementation of a sample server system, and show how the architecture can be scaled.

1 Rationale

In the future, we envision local networks serving thousands up to hundreds-of-thousands of resource-poor clients, e.g., NCs. These networks might be intra-building, intra-organization or even intra-city. Customizable servers will be required that nevertheless offer extremely high performance.

The low cost and variety of future clients (e.g., PDAs, laptops, pagers, printers, and specialized appliances) will result in a larger number of client devices per user. Each office employee could have tens of client devices. Thin clients will have fast processors, but little or no disk storage so that they will download most of their data and executables. Some typical applications for these clients will also download very large objects, such as graphics and video.

The existence of cheap client hardware with high-resolution graphics and high-quality audio together with ubiquitous high-bandwidth networks will probably lead to applications with increasing demands on network and server performance. For a scenario with 10,000 users and 100,000 thin clients, we think that requirements to the server like "handle 20,000 requests in a second with a data bandwidth of 1 GByte/s" will be realistic. Perhaps even higher bandwidth will be requested. In the near future, we envision clusters of 1,000 up to 2,000 NCs.

The postulated server performance is about two orders of magnitude higher than current servers achieve [10, 15]. We are convinced that improving current systems by evolution is not sufficient: we need a new server architecture to achieve the mentioned goals. The basic requirements to this architecture are *customizability, performance, scalability* and *security support*.

As Kaashoek *et al.* have noted recently [10], traditional servers are designed either to run a variety of applications, but with abstractions that lead to poor performance, or run specialized applications efficiently, but without the flexibility to run other applications. They define a *server operating system* environment that is designed to provide abstractions for building specialized, high-performance servers. While their server operating system enables improvements in server performance and flexibility, we claim that further improvement in performance is necessary and possible without reducing the variety of server systems that can be developed.

We believe that in our envisioned scenarios, significant performance improvements are possible by providing clients with access to local servers that optimize cache response. For example, an organization could use a central server (or cluster of servers) and 1000 NCs, all connected by a local area network. The NCs boot from the central server, use it as a file system, as a Web proxy, as a server for organization-internal HTML documents, and perhaps also for video clips. Some objects the server deals with will come from the Web; however most objects will be local to the organization (software, forms,

brochures, diagrams, custom data, etc.) so we expect a large but bounded working set.

In this scenario, the important problems are actually server latency and throughput, rather than network latency (as addressed by Web caching [3]). Network bandwidth for the central server to communicate with the clients is easily obtained (e.g., using multiple 100 Mbps Ethernets), so the problem in this scenario is to improve server performance such that it can utilize this bandwidth effectively.

We are aware of two principal ways for increasing a system's performance substantially beyond the bare performance growth of hardware: replication and caching. Massive replication of servers (e.g., IBM's Olympic server) is probably too expensive, makes write accesses complicated and slow, and needs sophisticated load balancing.

Therefore, we focus on the development of a high-performance, cache-based architecture that is general enough to support most type of server applications. Such an architecture should enable the server to achieve very close to the maximum performance that the architecture can achieve in principle for the fast path (i.e., hits on NC client requests in the local server's cache). In addition, customizability and handling heterogeneous objects are also relevant to the architecture because it must be general enough to support a wide variety of applications.

Our key decision for constructing high-performance customizable servers is to separate *generic cache modules* and *customizable miss-handling modules* and to map them to dedicated machines. Generic cache modules are responsible for high performance while customizable miss handlers enable flexibility. Single or multiple generic and customizable modules together build a general or specialized server (or server cluster). In the prototype, we use an off-the-shelf PC equipped with a 200 MHz Pentium Pro processor for a generic cache module.

In this paper, we focus on the generic cache module which is the centerpiece of the architecture. For it, we envision main-memory caches of 4 GB up to 64 GB. The challenge is to construct software that efficiently maintains such a cache, that does not restrict customizability, that supports scaling of multiple modules, and that is nevertheless highly specialized and optimized to achieve the demanded high throughput.

The proposed architecture heavily relies on the inherent "cache friendliness" of applications. Our hope is that due to the customizability of the architecture and due to the support of active objects in the generic cache module, most applications can benefit from the cache structure. However, we are still far from substantiating this hope. Currently, we have only implemented two small prototypes, an instructional video-clip server (see Section 2.1) and a web proxy. Using only a single custom module and a single generic cache module, the video-clip server is able to serve up to 402 clients with video clips simultaneously (MPEG I, 1.5 Mbps, full screen presentation). The performance is currently limited by the sub-optimal DMA of current PCI chipsets and memory buses.

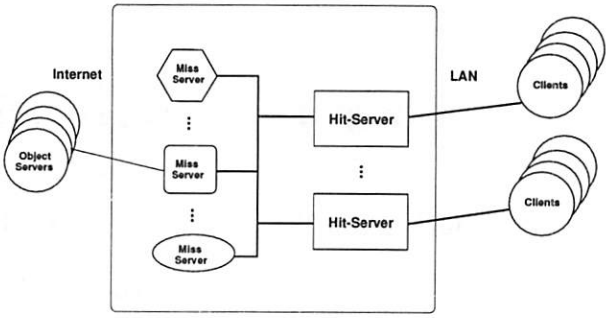


Figure 1: Server Architecture. A customized server has modules of two different types: hit-server(s) and miss-server(s). They are dedicated machines and are connected with a dedicated intra-server network. Client libraries reside on each client and are responsible for communicating with the hit-server via a LAN.

In Section 2, we describe the design of the server architecture, focused on the generic cache module. We detail the implementation of the hit-server in Section 3 and present performance results in Section 4. In Section 5, we discuss the scalability of the presented architecture. We review related work in Section 6 and conclude in Section 7.

2 The Server Architecture

The server architecture consists of two types of modules that cooperate to manage a large RAM cache of objects (shown in Figure 1). A *hit-server* is a highly-optimized generic module (a dedicated machine) that handles client requests that hit in the object cache. *Miss-servers* handle client requests that miss in the object cache. Miss-servers also implement application-specific policies for managing the hit-server cache and for distributing objects to clients. Multiple, different miss-servers can be combined with a single hit-server or with multiple hit-servers (see Section 5).

Clients read and write objects by performing *get* and *put* operations on server objects. For example, HTTP's *get* and *put* are mapped to Lava's *get* and *put* by client libraries. The Lava operations also work on partial objects. This feature permits clients to download or modify arbitrary, selected parts of the object (i.e., as opposed to reading the entire object). As well, client libraries can implement file-system like access. Furthermore, objects can have object-specific *get* and *put* operations supplied by the object creator. For example, a custom *get* can

present an object in different formats based on the requesting client. Another example is HTTP *post*: A combined *putget* operation sends the *post* data to the active object which then calculates the response and sends it back to the client as *get* data.

In fact, *putget* is the only existing operation from the hit-server's point of view. The *put* data is sent to the object which then delivers the *get* data. Pure *get* and *put* operations are implemented by *putget* using empty put or get parameters respectively. For better intuitive understanding, we will, however, always refer to pure *get* and *put* operations in the following sections.

In the remainder of this section, we develop the design of the server architecture. First, we describe an example server which demonstrates the interplay between the architecture components. Next, we analyze the application scenario to determine the requirements of the architecture. The last two subsections develop the design.

2.1 An Example Server

We implemented a video-clip server that delivers video clips interactively to clients. Examples of such a system include museum kiosks, retail services (e.g., mall, information resources), educational services (e.g., library and encyclopedia), and entertainment services. For instance, upon entering a museum, information about exhibits, resources, and staff can be retrieved from computers located at kiosks throughout the museum.

A prototype version of the video-clip server system is built using Windows 95 PC clients running the ActiveMovie application to view videos and a miss-server that runs on Linux 2.0 to retrieve the videos. Our instructional server provides small video clips (varying from about 20 seconds and 4 MB to about 180 seconds and 50 MB each) to its clients.

The Windows 95 clients use Lava's reliable object protocol to communicate with the hit-server. The protocol is implemented as a Windows kernel extension (a so-called Vxd element) that communicates directly with the network driver using Window's NDIS network-driver interface. A special ActiveMovie source filter was built that transfers ActiveMovie requests into Lava's *get* transactions. The ActiveMovie source filter requests a series of video blocks that correspond to a consecutive intervals of the video. The size of each block is 32 KBytes.

The according miss-server executes as a user process on Linux. Lava's reliable object protocol (see Section 2.4) is incorporated into the Linux kernel to enable hit-server/miss-server communication. Application-specific policy is added to: (1) download video objects into the miss-server from the Web using (unmodified) HTTP and (2) implement a custom cache-replacement policy that controls the hit-server.

In conjunction with the mentioned video-clip miss-server, the hit-server can serve up to 402 clients simultaneously with different video clips (see Section 4).

2.2 Requirement Analysis

The hit-server has an object cache that it uses to process *get* and *put* operations from network clients. Furthermore, it is also linked to miss-servers (e.g., web proxy, file system servers, and databases) from which it can obtain other objects to fulfill its clients' requests. Each hit-server communicates with its clients via network controllers that transfer data between the network and the server's main memory over the PCI bus and the memory bus. On the memory bus, CPU-memory and PCI-memory traffic compete with each other.

Our original goal (which turned out to be not completely achievable, see Section 4) was to build server systems whose server-to-client throughput rates approach the current PCI bus bandwidth of 1 Gbps. Utilizing this rate even for moderately small objects of 1 K, would require 128,000 transactions per second. For comparison: commercial servers currently achieve rates up to 2100 [15] transactions per second, research servers [11] up to 7000 transactions per second.

Since the memory bus is the bottleneck of a hit-server, the server architecture must maximize the availability of the memory bus to the network controllers.

In order to achieve high throughput and low latency, the server must also make efficient use of its object cache, basically a problem of deriving cache replacement and prefetching protocols that keep the "right" objects in the cache. We refer to applications in which such protocols can be defined as *cache-friendly*. In this paper, we make no claims about the design of such protocols (as others do, e.g., Cao et al. [2]). However, for applications where such protocols exist, the server architecture must permit their implementation. Also, the effect of processing cache misses on server throughput must be minimized.

Communication over untrusted links must be authenticated to prevent attacks. An authenticated communication is one whose source, integrity, and freshness have been verified. We do not believe that privacy is required for our server, at present. Certainly communication with object servers over the Internet needs to be authenticated. In addition, given the number of insider attacks reported and the value of corporate data, client communication over the LAN may also need to be authenticated. Therefore, the server architecture must enable the ability to authenticate communication along any link. However, we must minimize the effect that message authentication has on the hit-server's throughput rate.

The server systems must also be scalable through the addition of new server modules. Scalability is limited primarily by interactions caused by objects being written. When an object write occurs, consistency requirements of that object must be enforced. Many applications enforce a strict consistency in which a reader sees the latest writes. However, less restrictive policies, such as the various types of release consistency, are also used by distributed applications, so the server must support application-specific consistency policies.

In summary, the major requirements relate to four dimensions: *Performance, flexibility, security,* and *scalability.*

2.3 Hit-Server Architecture

The hit-server provides efficient mechanisms for processing client requests that hit in the object cache. Upon a client request, the hit-server locates the requested object and either downloads it to the client (on a *get*), creates a new version (on a *put*), or forwards the request to the miss-server (on a miss).

The hit-server is free of policy. Its general mechanisms are intended to support any policy that the miss-servers can implement, so developers can create application-specific server systems. For example, when cache replacement is signaled by the hit-server, the miss-server is free to select the objects to be replaced. A miss-server library provides general miss-server primitives and a set of functions that use these primitives to implement predefined policies. However, the developer can choose to build a miss-server from any combination of predefined and custom policies or even build a new miss server from scratch.

The hit-server processes client *get/put* requests to access a large RAM cache of objects and processes miss-server *add/remove* requests to modify the cache. Requests that result in cache misses are forwarded to the miss-server.

The default *get* operation first checks whether the object is available. Then, the client is authorized against the object's ACL. Next, the object's status data and consistency matrix may indicate that the object's miss-server be signaled, so it can implement the object's consistency policy (see Section 5 for details). Finally, after verifying that a download is necessary by checking version numbers, the hit-server sends the requested object data to the client.

The *put* operation is similar except that each *put* generates a new version of the object: First, the entire object is copied (lazily); then, those object parts are modified that are addressed by the *put* data. The mentioned copy operation is based on copy-on-write techniques; basically, the newly received packets are simply linked into the new version's data descriptor. (The per-object data descriptor is similar to a multi-level page table but implements a granularity of 1 K.) This technique makes it easy to update an object while *gets* are concurrently active. After all *gets* on the old version are finished, the old version can be garbage collected.

Similar to the DynamicWeb cache [8], the hit-server interface permits to cache dynamic Web pages or other dynamic objects. The application running on the miss server constructs the dynamic object on demand and invalidates or updates it in the hit-server whenever necessary. Note that this requires only default *putget* operations. If the method is too expensive, e.g. for a dynamic object that delivers random numbers, an object-specifc *putget* can be used that executes directly on the hit-server.

The architecture enables flexible handling of objects through object-specific *putget* operations. An object-specific operation supersedes the default. Object-specific *putget* methods are run on the hit-server to enable efficient implementation of custom operations. An object-specific *put* operation is invoked after the new data is received, but prior to updating the object cache. E.g., it can be used to implement object-specific consistency protocols that are executed when an update is made. An object-specific *get* operation is invoked prior to delivery to the client. An object-specific *get* can be used to deliver modified object data to a client (e.g., for displaying the object effectively on the client).

In order to prevent corruption of the hit-server and denial-of-service to clients, the hit-server must control these object-specific *putget* operations. We assign each object-specific *putget* operation to its own address space to protect the hit-server and other object-specific operations from modification. Resource requests, such as access to a client descriptor, are intercepted and authorized against the object-specific operations access rights [9, 13]. For example, object-specific operations are permitted to read the requesting client's description and the requested object descriptor. If multiple objects share the same *putget* operation, they are all mapped into the same address space. Address spaces are a relatively lightweight resource, see Section 3.2. Custom *putget* operations can be multithreaded to execute multiple requests concurrently.

The hit-server also processes miss-server operations, *add* and *remove*, by which miss-servers can add or remove objects (specified by name and version) from the object cache, respectively. An *add* enables the miss-server to set the initial values for the object's attributes.

2.4 Reliable Network Communication

Although we envision the use of switched networks, packet losses are possible. They can occur in switches or

even within the hit-server's Ethernet chips, even though the hit-server always has enough receive buffers available. The point of congestion is not main memory itself but the memory/PCI bus. As described in more detail in Section 3.1, the maximum memory-bus bandwidth for DMA is approximately 600 Mbps. As soon as the sum of all cards' incoming and outgoing Ethernet traffic exceeds[1] this value, the receiver FIFOs (4K each) in the Ethernet chips can run over. (The problem is real: the current hardware uses 7 full-duplex Ethernet cards enabling peaks of 1400 Mbps.)

The first obvious choice for a reliable transport protocol is TCP. Although some of its features are not necessary for our application (e.g., checksums, handling duplicates and out-of-order packets), adaptive flow control and retransmission of lost packets are required. It is well known that TCP is often costly in terms of processor cycles [18] and that a VMTP-like [4] protocol is better suited for transactions.

An even more important problem of TCP is that its congestion-avoidance policies (which are primarily based on end-to-end flow control) are tailored to current WANs and are not effective for our envisioned scenarios: On a highly loaded LAN, we experience dramatically changing loss rates, e.g. 0% loss for 5 ms, then 40% for 2 ms, etc. TCP would very quickly reduce its window size to a single packet. This would result in poor bandwidth utilization *and not avoid packet losses* since, in peak situations, the loss rate depends more on the synchronization between the clients than on the sender's transmission rate. Given that under peak load the round-trip time exceeds 1 ms (the client's and server's hardware FIFOs are even good for a 0.9 ms delay), it is very difficult to devise a flow-control protocol that can handle the described agility efficiently.

Instead, we use a late-retransmission protocol. Basically, any sender transmits the whole object in a burst, as fast as the network hardware permits. Afterwards, the receiver tells the sender which parts of the object it has received; finally, the sender retransmits the missing, i.e., lost, parts (if any). This procedure is repeated until all data is transferred.

The mentioned protocol does not work for any topology. However, it behaves nicely on a star topology as in our scenario where nearly all communication either goes to or comes from the hit-server. With a switched network, the protocol ensures that the hit-server receives data at its maximum rate and all clients get close to optimal bandwidth. At a first glance, this might be counterintuitive since the protocol seems to invite congestion rather than to avoid it. However, assume that two clients simultaneously send a 1 MByte object each to the same hit-server card. Due to congestion, always 1/2 of the data sent will be lost: On the first round, each client sends the entire object; on the second round, each client sends the lost 1/2 M, on the third the 1/4 M lost in the second round, etc. In total, each client sends 2 MByte with full speed to effectively transfer 1 MByte, i.e., gets 50% of the available bandwidth. In the same time, the hit-server receives 2×1 MByte at the highest possible rate. The point is that only such packets are lost that could not have been transmitted under ideal flow control, and that the "unnecessary" transmissions consume only resources that otherwise would be unused.[2]

Since this paper concentrates on the hit-server design, we will neither go into details of the protocol nor proof its properties nor discuss further "good" topologies here. For the context of this paper, it is relevant that the protocol is reliable, performs well under peak load, is cheap for low loss rates, is robust against random losses and highly fluctuating loss rates, and can be "asymmetrically" implemented such that it requires more processor cycles on the client side and less on the hit-server side.

3 Hit-Server Implementation

In this section, we describe the techniques used for implementing the generic hit-server. Miss-servers enable customizability and extensibility; the hit-server is responsible for performance. Consequently, its design is basically driven by performance requirements. In a first step, achievable performance goals are derived from the characteristics of the available hardware. Then, in an ideal-case micro analysis, we try to determine the load an optimal implementation would impose on processor, cache, memory bus, PCI bus and Ethernets. This analysis gives us a more realistic upper bound of the achievable throughput, and it reveals the bottlenecks of the system. Finally, guided by these results, we describe the actual construction of the hit-server core software.

[1] In reality, the situation is complicated by DMA bursts, bus arbitration policies and the existence of multi-level PCI buses. However, all this is hardware and most of its parameters and algorithms cannot be influenced by software. A detailed description is beyond the scope of this paper.

[2] There are some pathological situations. If, e.g., 100 clients simultaneously start sending an object of 100 packets, each round effectively transfers only 1 packet per client. Then we needed 1 ack per transferred packet, similar to 1-packet windows in TCP. (Nicely, we needed only 0.1 acks per transferred packet, if 1000-packet objects were sent.) Therefore, as soon as a client notices that the effective ratio of acks to effectively transferred packets becomes too high, it takes random rests while transmitting the packets. Since all active clients act in a similar way, the congestion and the loss rate decreases so that the ack ratios become better. For short objects, additional transmission rounds are needed: the packets are retransmitted a second or third time without waiting for an ack.

3.1 Analysis

Our current hit-server machine is an off-the-shelf PC, equipped with a 200-MHz Pentium Pro uniprocessor, an Intel 440 FX chipset, and 256-K of L2 cache memory. For our experiments, the hit-server was equipped with 256 M of main memory. External devices are connected to the processor and the memory by a 32-bit PCI bus with a cycle time of 30 ns (33 MHz). The PCI-bus specification [17] permits burst DMA transfers with a rate of 1 word per PCI-bus cycle, corresponding to 132 MByte/s or 1056 Mbps. However, the 440 FX chipset, at least in combination with the Ethernet controller chips we use, takes on average 1.5 cycles to transfer a word. So the maximum achievable transfer rate is 88 MByte/s or 704 Mbps.

The SMC EtherPower 10/100 PCI network cards we use support 100 Mbps Ethernets. They are based on the DEC 21140 AE ("Tulip") controller chip. Since the machine has only 4 PCI slots on its motherboard, we had to use an additional PCI bridge (DEC 21152) for connecting 7 Ethernet cards. In our experimental setup (Figure 2), 6 Ethernets are used as client networks, 4 of them are connected with the motherboard through the additional bridge. The seventh Ethernet connects the hit-server with the miss-servers.

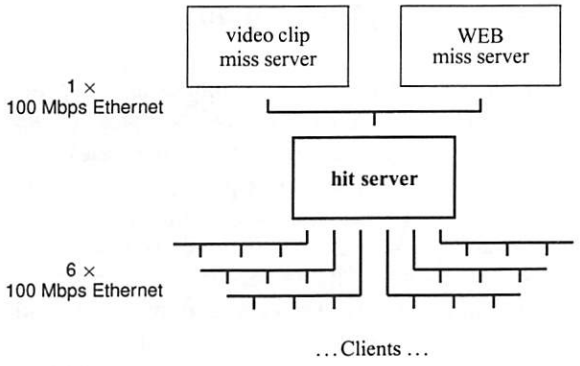


Figure 2: Single-Hit-Server Architecture.

For increased numbers of miss-servers and hit-servers in a server cluster, the inter-server network hardware can be upgraded: multiple Ethernets for point-to-point connections, an ATM switch or a Myrinet. Since the inter-server network connects only 2 to perhaps 15 nodes, the related costs are economically feasible.

Pre-Implementation Performance Analysis

From the performance point of view, the most relevant operations are delivering objects to clients and receiving requests. We start with an idealistic and optimistic *pre-implementation analysis* of these both basic functions. The purpose of this analysis is twofold: (1) estimate an upper bound of the achievable performance; (2) identify the system's potential bottlenecks. Of course, the thus determined idealistic performance is in practice not completely achievable. Nevertheless, it gives us a reasonable order-of-magnitude goal and helps us to concentrate on the relevant optimizations in the design. Furthermore, this methodology helps us checking whether the theory, i.e., our understanding of the system, is in accordance with the reality of the system. If later performance experiments roughly corroborate with the idealistic predictions, we have a certain confidence about theory and implementation. If experiments largely diverge with our theory, we either have the wrong model or made mistakes in implementing it.

Even if an ideal implementation of the hit-server core would spend no time for bookkeeping and OS overhead, sending and receiving packets through the Ethernet controller are unavoidable. So we first analyze the optimal costs for sending a packet. Figures 3 and 4 illustrate the interaction between processor and Ethernet controller.

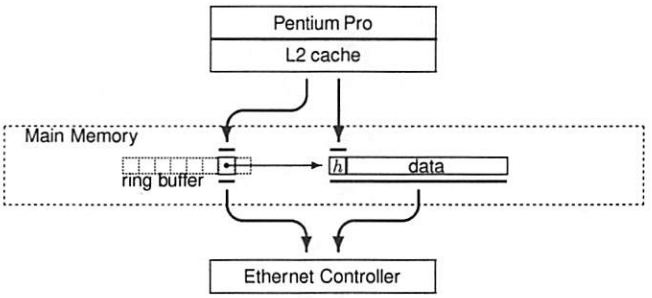


Figure 3: *Sending an Ethernet packet.* The *ring buffer* holds descriptors pointing (thin arrow) to the packets that the Ethernet controller should transmit. For each packet, the processor first writes the descriptor and the packet header; then the Ethernet controller reads the descriptor and the whole packet. Memory reads and writes are denoted by thick arrows.

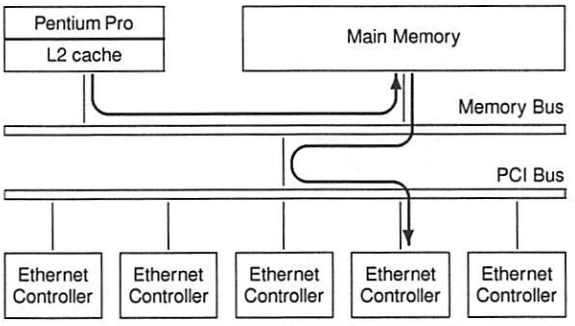


Figure 4: *Accessing Main Memory.* Processor read/writes use only the memory bus while transferring data to or from Ethernet controllers involves PCI bus and memory bus.

Both components communicate via the main memory:

the processor accesses the main memory through the memory bus and the Ethernet controller through the PCI bus *and* the memory bus.

For this analysis, we assume that the packets consist of 32 B header information and 1024 B object data. Transmitting a packet requires the following steps:

1. The system has to receive a device interrupt that indicates that the controller is ready to accept another packet. (Alternatively, the system has to poll the controllers, i.e., needs to read their status registers.) To generate the interrupt, the controller sends one word through the PCI bus. Furthermore, it writes its current status word, another PCI-bus write operation.
2. To ensure that no errors are pending, the system has to read the controller status (see step 1) from memory. Note that this is always an L2-cache miss, since the controller writes to memory and not into the L2-cache.
3. The system has to select a new packet. Under ideal assumptions, no L2-cache miss occurs for this. The main memory is not accessed.
4. The system has to prepare the new packet for transmission. This includes at least writing the client's Ethernet/IP address into the packet header: one cache line has to be written back.
5. The transmission has to be set up. For this purpose, the descriptor in the ring buffer has to be written. It needs at least the physical memory address of the new packet: one memory access. Furthermore, the Ethernet controller needs to be triggered for starting the transmission (one PCI word). Afterwards, the controller will read the according descriptor from the ring buffer, i.e. from memory: 4 words through the PCI bus.
6. Finally, the controller will transfer the packet from memory to its own bus: 256+8 words through the PCI bus.

In Table 1, the costs of these six steps are estimated and given for the critical components: processor, memory bus, PCI bus and Ethernet buses. Due to buffering and pipelining, these components can to a large degree work in parallel. However, main-memory reads through the PCI bus always require corresponding memory-bus activity.

Send Packet (1056 bytes)						
	processor [μs]	*lines by cpu*	Memory [μs]	*words by PCI*	PCI [μs]	7× Ethernet [μs]
1) device interrupt	1.90	–	0.30	2	0.48	–
2) inspect controller	0.40	1	0.29	–	–	–
3) select packet	0.10	–	–	–	–	–
4) prepare packet	0.57	1	0.29	–	–	–
5) setup transmission	0.72	1	0.57	5	0.36	–
6) transmit packet	–	–	12.09	264	12.09	84/7
total	3.69		**13.53**		12.45	12.00
utilization	27%		100%		92%	89%
max achievable rate: $\frac{1056 \times 8 \text{ bits}}{13.53\ \mu s}$ = **624 Mbps**						

Table 1: *Pre-Implementation Micro Analysis for a Hit-Server.* Processor costs are derived from instruction estimates (disregarding memory costs) and from micro benchmarks of the underlying μ-kernel. Memory and PCI-bus costs are calculated from the derived number of transfers and the average throughput costs of these transfer measured by micro benchmarks. Ethernet costs are derived from the specified throughput of 100 Mbps.

3.2 Implementation Techniques

From the previous analysis, we know that every 14 μs a 1 K packet has to be sent to achieve maximum hardware utilization. However, we had to design the system for an even higher demand: with optimal PCI-bus DMA hardware (1 word per 30 ns cycle), a packet had to be transmitted every 8 μs.[3] Obviously, the hit-server software must be constructed carefully not to delay packet transmission. The following paragraphs discuss the methods we used to achieve these goals.

[3] From the above discussion, we know that transmitting a 1024 B packet and a 32 B header requires $(1+1)_{step1} + (1+4)_{step5} + (256+8)_{step6} = 271$ word transfers. So in the ideal situation, 1 word per 30 ns through the PCI bus and no delay by the memory bus, 217 × 30ns = 8.13μs.

Early Evaluation

Early evaluation is a technique for reducing the latency and improving the throughput of operations. If an operation is requested several times on the same object, it needs to be executed only once. If an operation can be executed either at a place or at a time when free resources are available, its costs are hidden.

Object precompilation ensures that only negligible computations or data transformations are required by the hit-server for delivering a cached object to a client. When loading the object into the hit-server cache, the miss-server partitions it into network packets, generates the appropriate header information and computes a client-independent digest for each packet. On sending a packet, only the destination address, sequence number and sometimes a message-authentication code have to be generated.

To meet our security requirements, any delivered data has to be secured by a client-specific message-authentication code which is also unique in time to prevent replay attacks. By using a client-specific secret key, the authentication code is calculated from the precompiled client-independent digest and a nonce.

Many research experiments show that avoiding unnecessary data copies improves performance,(e.g., Fbufs [6], Unet [19]). Since we use precompiled packets in the object cache, we can always use unbuffered transmission

for delivery. With *put* operations, the arriving 1 K packets are linked, not copied, into the new version of the object (see also Section 2.3).

Per-Object Address Spaces

The default *putget* operation is implemented by the hit-server in a single address space. Since we would like to enforce security requirements on active objects, they execute their specialized operations in per-object address spaces.

Remember that the object granularity for *put* updates is 1 K while hardware pages are 4 K. Therefore, once a 4 K region of an object with a non-default *putget* operation is no longer physically contiguous, the corresponding page must be removed from the object's address space. Upon the page fault on this page, the corresponding 1 K chunks are physically copied into a fresh 4 K page frame which is then mapped into the object's address space. Object-specific *putgets* often do not read the entire object data by itself but simply specify to the hit-server core what parts should be transmitted. In these cases, the above mentioned lazy-copying technique avoids copying of received *put* data packets even for non-default objects.

The μ-kernel can be configured to support up to 65,000 address spaces. The space costs per address space are low for small objects, about 22 bytes for an object of 16 K. Nevertheless, the maximum number of address spaces is only 6.5% of the maximum number of objects. Currently, we do not yet know whether this is in practice sufficient to preallocate and preconstruct an address space for any object that uses non-default *get* and *put* operations. Otherwise, the hit-server would have to multiplex address spaces for active objects.

Minimizing Memory Conflicts

In the hit-server case, copying data in main memory is not only "in principle avoidable" but belongs to the class of the most expensive operations. Section 3.1 illustrated that the memory bus is the time-critical bottleneck. *Therefore, processor accesses to main memory have to be minimized.* Since L1 and L2 caches use a write-back strategy, the processor's reads and writes are uncritical as long as they hit in the hardware cache and do not touch the memory bus.

Fortunately, early evaluation techniques, in particular object precompilation, prevents unnecessary memory-to-cache copies. Since we use a precalculated digest, there is no need for the default *get* operation to read the object data.

The code segments of the μ-kernel and the hit-server core are small enough so that their frequently used parts fit completely even into the L1 cache. The hit-server's memory bus is not burdened with handling instruction misses. For data, the situation is different:

1. *Client descriptors*, basically the client's secret key and some status information, are not expected to cause frequent L2-cache misses, since they need only 2 cache lines per client. 400 simultaneously active clients need 10% of the L2 cache.
2. *Hash and name table* serve to identify a requested object. Since they are large, most accesses will cause L2-cache misses. Finding a 100-byte name then requires to read 1 line of the hash table and 4 lines of the name table, provided there is no name-hash conflict.
3. *Object descriptors* will frequently miss the L2 cache. Due to the large number of objects, we generally assume that the object-descriptor data is never found in the L2-cache upon a *get* request. Applications with many hot-spot objects will perform slightly better. Memory accesses are minimized by keeping object descriptors small:
   (a) The *object root* holds pointers to the object-specific operation, the object-page descriptor list and the object's ACL. 2 cache lines are accessed per request, one from the ACL and one to find the packet descriptors.
   (b) Any *object-page descriptor* contains the pointer to 4 packets forming the page and the corresponding 4 precalculated digest values. Together with links and a length field (objects can be smaller than a page) this fits into one cache line.
4. *Object data* is never read by default *get* operations. So no L2-cache misses occur in this case. Writing an object using a *put* operation requires message-authentication codes of the received data to be verified. This costs $1024/32 = 32$ cache misses per 1 K packet. (Checking the authentication code is omitted if the packet comes from a miss-server, since miss-servers are trusted and the inter-server interconnection is a closed network.)
5. *Packet headers* have to be constructed per packet transmission. For our hardware, packet header and the buffer descriptor required for the Ethernet controller together fit into one cache line. Since the Ethernet controller always transfers data from/to main memory, one L2-cache-line write and one read is required per transmission.
6. *Request data* is placed in main memory by the receiving Ethernet controller. Obviously, a request packet has to be read by the hit-server. For requesting an object with a 100-byte name, 4 cache lines have to be transferred.

Therefore in total, a default *get* on an n K object requires at least $11 + \lceil 9n/4 \rceil$ cache-line transfers between L2 cache and main memory.

4 Performance

In two throughput experiments, *get* requests for objects of 10 K and 1 K size are generated at the highest possible rate. A single physical client sends requests for randomly chosen virtual clients. (The Ethernet driver and the hit-server software are constructed such that, provided the utilized bandwidth does not exceed that of one card, there is no measurable difference between the packets coming through a single or through multiple cards.) To ensure an infinite burst, the request generator does not wait for a request to be completed. Avoiding the request-generation problems mentioned in [1], this method is good for generating up to 161,000 requests per second. To be sure to saturate the hit-server in a realistic way, we send requests with random gaps such that the hit-server gets slightly more requests on average than it completes. For the experiments, all objects were resident in the hit-server so that no miss-server communication was required.

For 10 K objects, we achieve a throughput of 594 Mbps with the current hardware, 7,000 transactions per second. For 1 K objects, the bandwidth is 304 Mbps, approximately 36,000 transactions per second. Note that all delivered data is authenticated. Corroborating these results, our video-clip server can serve up to 402 clients with different MPEG I video clips (1.5 Mbps) simultaneously.

The crucial question is of course whether our approach performs significantly better than conventional server architectures. Therefore, we compare Lava with some other research systems and commercial Web servers *in our envisioned LAN scenario* (many NCs in a local cluster). The reader should note that the results cannot simply be extrapolated to other application fields, e.g., wide-area networks. In particular, the systems are not functionally equivalent: e.g., the commercial Web servers support TCP clients but cannot be customized like the Lava architecture.

Figure 5 illustrates that for our LAN-based application Lava's hit-server offers an order-of-magnitude increase over conventional systems. We report net data throughput, i.e., do not count headers etc.

The numbers for Harvest [3] and Cheetah are taken from [11] and converted to Mbps. Both systems run on a 200 MHz Pentium Pro machine like the Lava hit-server. Harvest runs on top of the BSD operating system, Cheetah on top of MIT's Xok system. IO-Lite [16] runs on a 233 MHz Alpha station with two 100 Mbps network adapters. For comparison with industry-standard servers, we include numbers for Microsoft's Internet Information Server 4.0 running on a 166 MHz Pentium with Windows NT 4.0. Afpa [15] is an NT-based server running on a Pentium 166 processor. Both the MIIS and Afpa peformance have been measured in our lab by eliminating all client and network bottlenecks and increasing load until server was saturated. Similar to our experiment, the measurements for Cheetah, Harvest, MIIS, and Afpa are based on a LAN with no competing traffic; *get* requests are always served from the systems' respective main-memory caches. All systems implement the HTTP functionality; for Lava however, the client library translates HTTP requests to object-protocol requests on the client side.

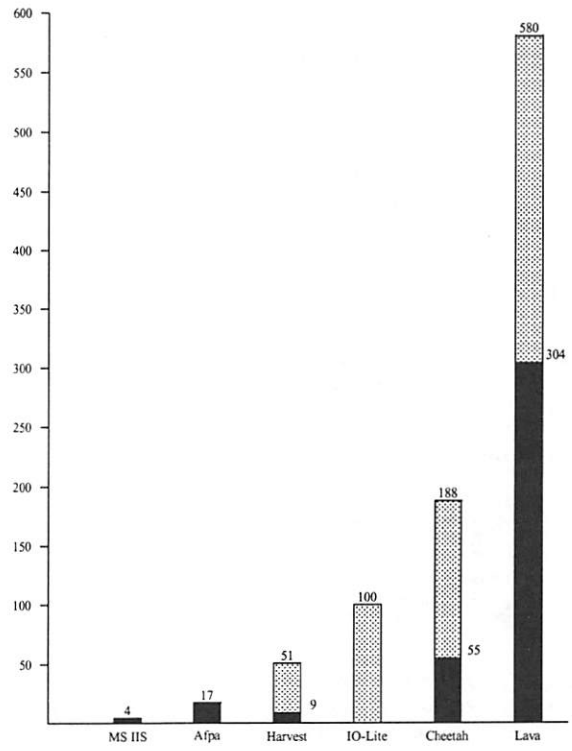


Figure 5: *Server Throughput.* Net data throughput in Mbps for get operations. (Headers, checksums, etc. are not considered to be net data.) Black bars denote the throughput for 1 K objects, shaded bars for 10 K objects. All systems are measured on a local area network and deliver data from their respective main-memory object cache.

Based on the get-throughput experiment, we simulated a system where a hit-server is used as a boot server for 1000 NCs. For booting, each NC had to download an individual set of objects, together 10 Mbyte per NC. We assumed that all 1000 NCs are turned on within the same 5-minute interval, equally distributed over time. We further assumed that each NC, once it is booted, starts working and then every second *get*s a 20 K object. When a user turns on her/his NC, how long would (s)he have to wait until the 10 M of boot data are downloaded? We found an average boot latency of 1.7 s with a standard deviation of 0.9 s (see Figure 6).

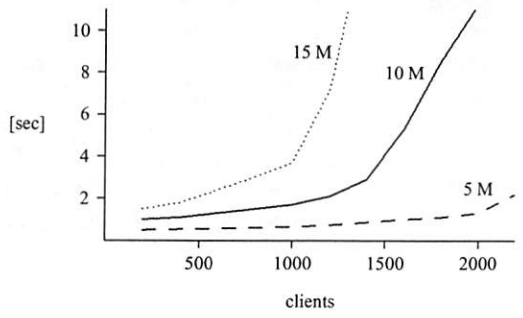


Figure 6: *Average Boot Latency.* All clients boot within the same 5-minute interval. Boot data is client-specific but of equal size for all clients (5 M, 10 M, or 15 M).

Miss handling does not substantially degrade the hit-server's throughput, since most of the work (loading the object) has to be done by the miss-server. Only during transferring an object from the miss-server into the hit-server, the hit-server's delivery rate decreases by 18%, basically because the miss-server communication consumes 100 Mbps from the total transfer bandwidth. (Transferring a 1 M object takes approximately 0.1 s.) Table 2 shows miss costs relative to hits for a single-disk Linux file system used as miss-server.

	1 K	10 K	100 K	1 M
miss latency	20 ms	21 ms	35 ms	162 ms
hit latency	0.1 ms	0.9 ms	9 ms	86 ms
hit : miss latency	1 : 200	1 : 23	1 : 3.9	1 : 1.9
hit : miss bandwidth	1200 : 1	140 : 1	22 : 1	10 : 1

Table 2: *Hit/miss costs.* The miss server file system runs on a 166 MHz Pentium with a Caviar 33100 disk. All data reflect ideal situations in which requests are not delayed by competing requests. Congestion at the miss-server or at a hit-server's Ethernet card would increase the latency. For the bandwidth ratios, we assume that the hit-server concurrently delivers objects of the same size on all cards.

For local networks, the throughput experiment gives some evidence that the Lava architecture enables an order-of-magnitude larger server/NC configurations than conventional server architectures. Whether the architecture can be modified to work efficiently in a wide area network is an open research problem.

5 Scalability

For many applications, a single hit-server might support up to 1000 clients. Future 66 MHz-PCI devices and 100 MHz memory buses might perhaps enable twice as many clients. However, for achieving our original goal of more than 10,000 clients, we must scale hit-servers. Miss-servers and hit-servers may also be scaled to reduce the miss latency or increase the effective cache size.

5.1 Adding Server Modules

There are three methods for adding server modules: (a) add miss-servers either to decrease miss latency or to handle heterogeneous objects, (b) add hit-servers to increase the cache size and hit rate, and (c) add hit-servers to improve the total bandwidth and handle more clients.

(a) Scalability is aided by an explicit separation of miss-server and hit-server hardware: miss-server CPU and IO consumption does not degrade hit-server throughput. Miss handling only influences the throughput of the hit-server when the miss-server stores an object into the hit-server cache.

(b) For certain cases, the bandwidth of a single hit-server might be sufficient but its main-memory cache might be too small for the application's working set. In particular, this can happen if the hit-server's motherboard supports less memory than the processor can address. Then, multiple hit-servers can be used to increase the object cache. Each hit-server holds the entire cache directory but the cached objects are partioned among all hit-servers. Client requests are multicasted to all hit-servers. If the request hits, the according hit-server executes it; the other ones classify the request as a hit but ignore it since they do not have the object. If a request misses, all hit-servers detect a miss and a dedicated hit-server signals it to the miss-server. This one then selects a hit-server for loading the new object. Fortunately, no complicated consistency protocol is required for this type of scaling. Network load, miss-server load and hit-server bandwidth are identical to the single hit-server case; only the resulting cache size is increased.

(c) When the number of clients becomes too large, hit-servers must be scaled to increase the total bandwidth of the system. This is simple as long as all objects are read-only. As soon as objects are write-shared between multiple hit-servers, we need consistency protocols.

5.2 Consistency

For sake of customizability and extensibility, the hit-server provides a consistency *mechanism* from which *per-object* consistency protocols can be implemented by miss-servers. So policies can be fully customized.

Unlike a hardware bus, a LAN does not enable snooping-based solutions for coherency. Instead, we enable the use of miss-servers as arbiters that can coordinate conflicting accesses to objects that are shared by multiple hit-servers.

The hit-server provides a single basic consistency mechanism: per-object *consistency-action matrices.* Two status bits are managed per object: *accessed* is set for any operation, *dirty* is set when a *put* operation occurs. The hit-server never resets these bits. The miss-

server, however, can arbitrarily change them. Combination of the four states with the two possible operations *(get/put)* leads to a 2×4 consistency-action matrix. Miss-servers specify a consistency action for each of the 8 fields for each object (in an object descriptor's consistency-matrix attribute). Four different consistency actions are available:

Ignore. The *get/put* operation takes place without involving the miss-server.

Notify. The object's miss-server is notified about the *get/put* operation. This notification is non-blocking. The miss-server will be informed concurrently to serving the client's request.

Call. The object's miss-server is called before the *get/put* request is served. The request blocks until the miss-server grants or denies it. In its reply, the miss-server can define new settings for the object's *accessed/dirty* bits and the consistency-action matrix. Before replying to a call, the miss-server can itself read the object back or update its value. Call-associated actions are completely controlled by the corresponding miss-server, and have a higher latency than ignore-associated and notify-associated actions.

Propagate. Any *put* operation is directly propagated to the corresponding miss-server (*put-through*). The data received from the client is sent to the miss-server and concurrently used for updating the object in the hit-server. Receive and propagate activities are pipelined; however, the client is not acknowledged until the miss-server commits or aborts the operation. Prior to handling a client's *get* request, the hit-server itself "*gets*" the newest version of the object from the corresponding miss-server (*get-through*). (*Get* requests include no data transfer if the requestor already holds the current object version.) To minimize the latency, receiving the new object data from the miss-server and propagating it to the client overlap in time.

The consistency-action matrix is a generic mechanism that can be used to implement a variety of different cache-consistency protocols and also cache-replacement protocols.

For a simple *write-through* policy, *(i,i,i,i / p,p,p,p)*[4] could be used: get operations on this object do not involve the miss-server whereas any put operation is directly propagated. For implementing *write-back* together with LRU replacement, the miss-server can use *(n,i,n, i / n,n,n,i)* . Then the miss-server is notified (a) when the object is accessed the first time (read or write) and (b) when the object becomes dirty (first write). Subsequent accesses do not involve miss-server notification. For LRU bookeeping, the miss-server will periodically reset all objects to *unaccessed* that have been accessed in the meantime. New accesses are then signaled (once per period) to the miss server. Inactive objects do not incur bookeeping overhead since there is no need for the miss-server to scan them periodically.

For consistency in a multi-hit-server system with write-shared objects, a MESI-like policy can be used:

(i,i,i,i / i,i,i, i)	*(i,i,i,i / c,c,c, c)*	*(c,c,c,c / c,c,c, c)*
for **M** or **E** objets	for **S** objects	for **I** objects

Modified (M) and exclusive (E) objects reside only in one hit-server. Clean objects are called E, dirty ones are called M. Accessing an M or E object does not involve miss-server activity. Shared (S) objects can reside in multiple hit-servers: *get*s happen without involving the miss-server whereas *put*s invoke a *call* consistency action. As a result of the *call*, the miss-server can invalidate the replicas of the according object in all other hit servers by changing their respective consistency-action matrices or even by removing them completely. For classical MESI, the miss-server will afterwards change the original object's consistency-action matrix to the M state and permit the *put* operation. Invalid (I) objects *call* the miss-server for any access so that this one could update the object replica and permit the access or simply delay it by delaying its reply.

6 Related Work

Internet caching has attracted a substantial amount of work. Web proxy caches, e.g. the original CERN web cache [14], are client-oriented, while hierarchical internet caches like Harvest/Squid [3] aim at reducing both backbone traffic and end-to-end network latency. As Kroeger *et al.* [12] and Duska *et al.* [7] report, Web hit-rates on a wide-area network are only about 40% to 50%; latency can be reduced by 30% to 60%.

We use caching for a completely different purpose. Instead of reducing wide-area traffic and network latency, we aim at improving server latency and server throughput. In a way, that is similar to Iyengar and Challenger [8] who concentrate on how to use caches on a server to improve the generation of dynamic Web pages.

Server operating systems have been discussed recently by Kaashoek et al. [10]. Cheetah's design exploits the underlying characteristics of the Exokernel [11] to construct servers that can access the hardware at low overheads. Cheetah uses kernel extensions to achieve high performance. Our hit-server runs entirely at user level. In some sense, our active objects are similar to ASHes [20] but are geared to multimedia documents, provide an object-oriented structuring technique, run entirely at user level and use supervised IPC.

The facilities we used to securely execute the custom methods of active objects could be used securely support ActiveIP [21]. In this sense, our techniques could be used to build an extremely fast router. One key difference between activeIP and our techniques is that since we use

[4] We denote a consistency-action matrix by *(get-clean-unacc, get-clean-acc, get-dirty-unacc, get-dirty-acc / put-clean-unacc, put-clean-acc, put-dirty-unacc, put-dirty-acc)*, where the first four entries specify the actions for get operations, the second row for put operations. Consistency actions are qualified by their first letter, *i, n, c,* and *p*.

hardware based protection and fast authorized IPC, our code does not need to be interpreted to be supervised but regular binaries can be used instead.

ADC [5] and Fbufs[6] are techniques used to improve the performance of network protocols and drivers for high speed networks and focus on reducing the number of data copies. We use similar techniques but go beyond them as we concentrate more on server throughput and network scheduling, rather than point-to-point network protocol throughput.

7 Conclusions

We have described the centerpiece of a server architecture for designing high-performance LAN servers. The server is separated into generic cache modules and custom modules. The generic module is policy-free and implements general and optimized mechanisms to handle cache requests. The custom modules can enforce arbitrary policies on the management and use of the cache (including authentication and object consistency). For cache-friendly applications, the resulting servers can perform an order-of-magnitude faster than existing systems. Since the custom modules can implement application-specific cache management policies, the likelihood that an application is cache-friendly can be increased.

We have learned that reducing main memory conflicts between the CPU and the network conrollers is the key to achieve high network throughput. Therefore, we provide a detailed examination into how the cache module must be designed to make efficient use of all hardware components.

There remain many open questions about how to use this server architecture. It is difficult to determine the cache-friendliness of applications designed to support thousands of clients in a laboratory setting. In the future, we plan to investigate the breadth of applicability of this architecture to current applications and investigate new classes of applications that may be enabled by our architecture.

Acknowledgements

We thank Rich Neves for the Afpa and MIIS measurement data and Yoonho Park for multiple discussions. We furthermore appreciate the comments of the anonymous reviewers and Mike Schwartz, our shepherd.

References

[1] G. Banga and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, pages 61–71, Monterey, CA, December 1997.

[2] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-contolled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel. A hierarchical internet object cache. In *1996 USENIX Technical Conference*, pages 153–163, January 1996.

[4] D. Cheriton. VMTP versatile message transaction protocol. RFC 1045, NRL, February 1988.

[5] P. Druschel, L. Peterson, and B. Davie. Experiments with a high-speed network adaptor: A software perspective. In *SIGCOMM '94 Conference*, 1994.

[6] Peter Druschel and Larry L. Petersen. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville NC (USA), December 1993. ACM.

[7] B. M. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *USENIX Symposium on Internet Technologies and Systems*, pages 23–35, Monterey, CA, December 1997.

[8] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, pages 49–60, Monterey, CA, December 1997.

[9] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[10] F. Kaashoek, D. Engler, G. Ganger, and D. Wallach. Server operating systems. In *1996 SIGOPS European Workshop*, September 1996.

[11] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 52–65, St. Malo, October 1997.

[12] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, pages 13–22, Monterey, CA, December 1997.

[13] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[14] A. Luotonen, H. Frystyk, and T. Berners-Lee. CERN httpd. http://www.w3.org/Daemon/Status.html.

[15] R. Neves. Personal communication, March 1997.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. Technical Report CS TR97-294, Rice University, Houston, TX, 1997.

[17] PCI SIG, Hillsboro, OR. *PCI Specification, Rev. 2.1S*, August 1996.

[18] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *1997 USENIX Technical Conference*, pages 257–274, Anaheim, CA, January 1997.

[19] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain (USA), December 1995. ACM.

[20] D. Wallach, D. Engler, and F. Kaashoek. Ashs: Application-specific handlers for high-performance messaging. In *SIGCOMM '96 Conference*, August 1996.

[21] David Wetherall and David Tennenhouse. The Active IP Option. In *Proceedings of the 1996 SIGOPS European Workshop*. ACM, 1996.

Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet

*Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, and Kenneth G. Yocum**
Department of Computer Science
Duke University
{*anderson, chase, gadde, gallatin, grant*}*@cs.duke.edu*

Michael J. Feeley
Department of Computer Science
University of British Columbia
feeley@cs.ubc.ca

Abstract

Recent advances in I/O bus structures (e.g., PCI), high-speed networks, and fast, cheap disks have significantly expanded the I/O capacity of desktop-class systems. This paper describes a messaging system designed to deliver the potential of these advances for network storage systems including cluster file systems and network memory. We describe *gms_net*, an RPC-like kernel-kernel messaging system based on Trapeze, a new firmware program for Myrinet network interfaces. We show how the communication features of Trapeze and *gms_net* are used by the Global Memory Service (GMS), a kernel-based network memory system.

The paper focuses on support for zero-copy page migration in GMS/Trapeze using two RPC variants important for peer-peer distributed services: (1) *delegated RPC* in which a request is delegated to a third party, and (2) *nonblocking RPC* in which replies are processed from the Trapeze receive interrupt handler. We present measurements of sequential file access from network memory in the GMS/Trapeze prototype on a Myrinet/Alpha cluster, showing the bandwidth effects of file system interfaces and communication choices. GMS/Trapeze delivers a peak read bandwidth of 96 MB/s using memory-mapped file I/O.

1 Introduction

Two recent hardware advances boost the potential of cluster computing: switched cluster interconnects that can carry 1Gb/s or more of point-to-point bandwidth, and high-quality PCI bus implementations that can handle data streams at gigabit speeds. We are developing system facilities to realize the potential for high-speed data transfer over Myricom's 1.28 Gb/s Myrinet LAN [2], and harness it for cluster file systems, network memory systems, and other distributed OS services that cooperatively share data across the cluster. Our broad goal is to use the power of the network to "cheat" the I/O bottleneck for data-intensive computing on workstation clusters.

This paper describes use of the Trapeze messaging system [27, 5] for high-speed data transfer in a network memory system, the Global Memory Service (GMS) [14, 18]. Trapeze is a firmware program for Myrinet/PCI adapters, and an associated messaging library for DEC AlphaStations running Digital Unix 4.0 and Intel platforms running FreeBSD 2.2. Trapeze communication delivers the performance of the underlying I/O bus hardware, balancing low latency with high bandwidth. Since the Myrinet firmware is customer-loadable, any Myrinet network site with PCI-based machines can use Trapeze.

GMS [14] is a Unix kernel facility that manages the memories of cluster nodes as a shared, distributed page cache. GMS supports remote paging [8, 15] and cooperative caching [10] of file blocks and virtual memory pages, unified at a low level of the Digital Unix 4.0 kernel (a FreeBSD port is in progress). The purpose of GMS is to exploit high-speed networks to improve performance of data-intensive workloads by replacing disk activity with memory-memory transfers across the network whenever possible. The GMS mechanisms manage the movement of VM pages and file blocks between each node's *local page cache* — the file buffer cache and the set of resident virtual pages — and the network memory *global page cache*.

This paper deals with the communication mechanisms and network performance of GMS systems using Trapeze/Myrinet, with particular focus on the support for zero-copy read-ahead and write-behind of sequentially accessed files. Cluster file systems that stripe data across multiple servers are typically limited by

*This work is supported by the National Science Foundation under grants CCR-96-24857 and CDA-95-12356, equipment grants from Intel Corporation and Myricom, and a software grant from the Open Group.

the bandwidth of the network and communication system [23, 16, 1]. We measure synthetic bandwidth tests that access files in network memory, in order to determine the maximum bandwidth achievable through the file system interface by any network storage system using Trapeze. The current GMS/Trapeze prototype can read files from network memory at 96 MB/s on an AlphaStation/Myrinet network. Since these speeds approach the physical limits of the hardware, unnecessary overheads (e.g., copying) can have significant effects on performance. These overheads can occur in the file access interface as well as in the messaging system. We evaluate three file access interfaces, including two that use the Unix *mmap* system call to eliminate copying.

Central to GMS is an RPC-like messaging facility (*gms_net*) that works with the Trapeze interface to support the messaging patterns and block migration traffic characteristic of GMS and other network storage services. This includes a mix of asynchronous and request/response messaging (RPC) that is *peer-to-peer* in the sense that each "client" may also act as a "server". The support for RPC includes two variants important for network storage: (1) *delegated RPCs* in which requests are delegated to third parties, and (2) *nonblocking RPC* in which the replies are processed by *continuation* procedures executing from an interrupt handler. These features are important for peer-to-peer network storage services: the first supports directory lookups for fetched data, and the second supports lightweight asynchronous calls, which are useful for prefetching. When using these features, GMS and *gms_net* cooperate with Trapeze to unify buffering of migrated pages, eliminating all page copies by sending and receiving directly from the file buffer cache and local VM page cache.

This paper is organized as follows. Section 2 gives an overview of the Trapeze network interface and the features relevant to GMS communication. Section 3 deals with the *gms_net* messaging layer for Trapeze, focusing on the RPC variants and zero-copy handling of page transfers. Section 4 presents performance results from the GMS/Trapeze prototype. We conclude in Section 5.

2 High-Speed Data Transfer with Trapeze

The Trapeze messaging system consists of two components: a messaging library that is linked into programs using the package, and a firmware program that runs on the Myrinet network interface card (NIC). The Trapeze firmware and the host interact by exchanging commands and data through a block of memory on the NIC, which is addressable in the host's physical address space using programmed I/O. The firmware defines the interface between the host CPU and the network device; it interprets commands issued by the host and controls the movement of data between the host and the network link. The host accesses the network using macros and procedures in the Trapeze library, which defines the lowest level API for network communication across the Myrinet.

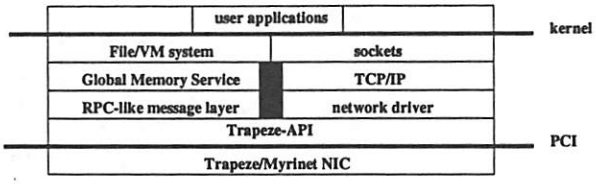


Figure 1: Using Trapeze for TCP/IP and for kernel-kernel messaging for network memory.

Like other network interfaces based on Myrinet (e.g., Hamlyn [4], VMMC-2 [13], Active Messages [9], FM [21]), Trapeze can be used as a memory-mapped network interface for user applications, e.g., parallel programs. However, Trapeze was designed primarily to support fast kernel-to-kernel messaging alongside conventional TCP/IP networking. The Trapeze distribution includes a network device driver that allows the native TCP/IP protocol stack to use a Trapeze network alongside the *gms_net* layer. Figure 1 depicts this structure. The kernel-to-kernel messaging layer is intended for GMS and other services that assume mutually trusting kernels.

2.1 Trapeze Overview

Trapeze messages are short *control messages* (maximum 128 bytes) with optional attached *payloads* typically containing application data not interpreted by the message system, e.g., a file block, a virtual memory page, or a TCP segment. Each message can have at most one payload attached to it. Separation of control messages and bulk data transfer is common to a large number of messaging systems since the V system [6].

A Trapeze control message and its payload (if any) are sent as a single packet on the network. Since Myrinet has no fixed maximum packet size (MTU), the maximum payload size of a Trapeze network is configurable, and is typically set to the virtual memory page size (4K or 8K). The Trapeze MTU is the maximum control message size plus the payload size.

Payloads are sent and received using DMA to/from aligned buffers residing anywhere in host memory. The host attaches a payload to an outgoing message using a Trapeze macro that stores the payload's DMA address and length into designated fields of the send ring entry. On the receiving side, Trapeze deposits the payload into a host memory buffer before delivering the control message.

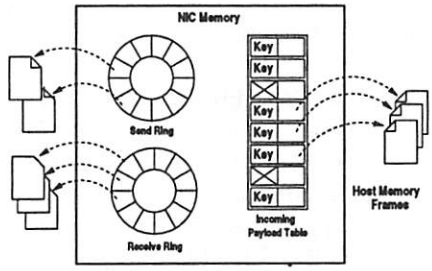


Figure 2: NIC Memory Structures for a Trapeze endpoint.

The data structures in NIC memory include an *endpoint* structure shared with the host. A Trapeze endpoint (shown in Figure 2) includes two message rings, one for sending and one for receiving. Each message ring is a circular array of 128-byte control message buffers and related state, managed as a producer/consumer queue. From the perspective of a host CPU, the NIC produces incoming messages in the receive ring and consumes outgoing messages in the send ring. The host sends a message by forming it in the next free send ring entry and setting a bit to indicate that the message is ready to send. When a message arrives from the network, the firmware deposits it into the next free receive ring entry, sets a bit to inform the host that the message is ready to consume, and optionally signals the host with an interrupt.

Handling of incoming messages is interrupt-driven when Trapeze is used from within the kernel. Each kernel protocol module using Trapeze (i.e., *gms_net* and the IP network driver) registers a receiver interrupt handler upcalled from the Trapeze interrupt handler.

Trapeze is designed to optimize handling of payloads as well as to deliver good performance for small messages. In a network memory system, page fault stall time is determined primarily by the time to transfer the requested page on the network. On the other hand, bursts of page transfers (e.g., for read-ahead for sequential access) require high bandwidth. The Trapeze firmware employs a message pipelining technique called *cut-through delivery* [27] to balance low payload latency with high bandwidth under load. With this technique, the one-way raw Trapeze latency for a 4K page transfer is 70μs on 300MHz Pentium-II/440LX systems with LANai 4.1 M2M-PCI32 Myrinet adapters. On these systems, Trapeze delivers 112 MB/s for a stream of 8K payloads; with 64K payloads, Trapeze can use over 95% of the peak bandwidth of the I/O bus, achieving 126 MB/s of user-to-user point-to-point bandwidth.[1]

[1]These bandwidth numbers define a "megabyte" as one million bytes. All other bandwidth numbers in this paper define 1MB as 1024*1024 bytes.

2.2 Unified Buffering for In-Kernel Trapeze

All kernel-based Trapeze protocol modules share a common pool of receive buffers allocated from the virtual memory page frame pool; the maximum payload size is set to the virtual memory page size. Since Digital Unix allocates its file block buffers from the virtual memory page frame pool as well, this allows unified buffering among the network, file, and VM systems. For example, the system can send any virtual memory page or cached file block out to the network by attaching it as a payload to an outgoing message. Similarly, every incoming payload is deposited in an aligned physical frame that can mapped into a user process or hashed into the file cache. Since file caching and virtual memory management are reasonably unified, we often refer to the two subsystems collectively as "the file/VM system", and use the term "page" to include file blocks.

The TCP/IP stack can also benefit from the unified buffering of Trapeze payloads to reduce copying overhead by *payload remapping* (similar to [11, 3, 17]). On a normal transmission, IP message data is copied from a user memory buffer into an mbuf chain [20] on the sending side; on the receiving side, the driver copies the header into a small mbuf, points a BSD-style external mbuf at the payload buffer, and passes the chain through the IP stack to the socket layer, which copies the payload into user memory and frees the kernel buffer. We have modified the Digital Unix socket layer to avoid copying when size and alignment properties allow. On the sending side, the socket layer builds mbuf chains by pinning the user buffer frames, marking them copy-on-write, referencing them with external mbufs, and passing them through the TCP/IP stack to the network driver, which attaches them to outgoing messages as payloads. On the receiving side, the socket layer unmaps the frames of the user buffer, replaces them with the kernel payload buffer frames, and frees the user frames. With payload remapping, AlphaStations running the standard *netperf* TCP benchmark over Trapeze sustain point-to-point bandwidth of 87 MB/s.[2]

Since outgoing payload frames attached to the send ring may be owned by the file/VM system, they must be protected from modification or reuse while a transmit is in progress. Trapeze notifies the system that it is safe to overwrite an outgoing frame by upcalling a specified *transmit completion handler* routine. For example, when an IP send on a user frame completes, Trapeze upcalls the completion routine, which unpins the frame and

[2]Measured Alcor (266 MHz AS 500) to Miata (500 MHz PWS 500au), 8320-byte MTU, 1M netperf transfers, socket buffers at 1M, software TCP checksums disabled (hardware CRC only): 732 Mb/s.

releases its copy-on-write protection.

However, to reduce overhead Trapeze does not generate transmit-complete interrupts. Instead, Trapeze saves the handler pointer in host memory and upcalls the handler only when the send ring entry is reused for another send. Since messages may be sent from interrupt handlers, a completion routine could be called in the context of an interrupt handler that happened to reuse the same send ring entry as the original message. For this reason, completion handlers must not block, and the structures they manipulate must be protected by disabling interrupts. Since completion upcalls may be arbitrarily delayed, the Trapeze API includes a routine to poll all pending transmits and call their handlers if they have completed.

2.3 Incoming Payload Table

The benefits of high-speed networking are easily overshadowed by processing costs and copying overhead in the hosts. To support zero-copy communication, a Trapeze receiver can designate a region of memory as the receive buffer space for a specific incoming payload identified by a tag field. When the message arrives, the firmware recognizes the tag and deposits the payload directly into the waiting buffer. Handling of tagged payloads is governed by a third structure in NIC memory, the *incoming payload table* (IPT).

GMS uses the Trapeze IPT for copy-free handling of fetched pages in RPC replies, as described in Section 3. Ordinarily, Trapeze payloads are received into buffers attached by the host to the receive ring entries; since the firmware places messages in the ring in the order they arrive, the host cannot know in advance which generic buffer will be selected to receive any given payload, and the payload may need to be copied within the host if it cannot be remapped. Early demultiplexing with the IPT avoids this copy.

To set up an IPT mapping, the host calls a Trapeze API routine to allocate a free entry in the IPT, initialize it with the DMA address of the designated payload buffer, and return a tag value (*payload token*) consisting of an IPT index and a protection key. The payload token is a weak form of capability that can be passed in a message to another node; any node that knows the token can use it to tag a message and transmit a payload into the buffer. When the firmware receives a tagged message from the network, it validates the key against the indexed IPT entry before initiating a DMA into the designated receive buffer. The receiving host may cancel the IPT entry at any time (e.g., request timeout); similarly, the firmware protects against dangling tokens and duplicate messages by cancelling the entry when a matching message is received. If the key is not valid, the NIC drops the payload and delivers the control portion with a payload length of zero, so the receive message handler can recognize and handle the error.

At present, the IPT maps only a few megabytes of host memory, enough for the reply payloads of all outstanding requests (e.g., outstanding page fetches). This is a modest approach that meets our needs, relative to more ambitious approaches that indirect through TLB-like structures on the NIC [13, 26, 7]. We have considered a larger IPT with support for multiple transfers to the same buffer at different offsets, as in Hamlyn's *sender-based memory management* [4], but we have not found a need for these features in our current uses of Trapeze.

3 Page Transfers in GMS/Trapeze

This section outlines a Trapeze-based kernel-kernel RPC-like messaging layer designed to support cooperative cluster services. The package is derived from the original RPC package for the Global Memory Service [14] (*gms_net*), extended to use Trapeze and to support a richer set of communication styles, primarily for asynchronous prefetching at high bandwidth [24]. Although the package is generic, we draw on GMS examples to motivate its features and to illustrate their use.

Since many aspects of RPC and messaging systems are well-understood, we focus on those aspects that benefit from the Trapeze features discussed in the previous section. In particular, we explain the features for transferring pages (or file blocks) efficiently within the RPC framework, and their use by the protocol operations most critical for GMS performance: page fetches (*getpage*) from the global page cache to a local page cache, and page pushes or evictions (*putpage* or *movepage*) from a local cache to the global cache.

Section 3.2 discusses the zero-copy handling of fetched pages using the Trapeze incoming payload table (IPT); Sections 3.3 and 3.4 extend the zero-copy reply scheme to delegated and nonblocking RPC variants useful in GMS and other peer-to-peer network services. We illustrate use of nonblocking RPC to extend standard read-ahead for files and virtual memory to GMS; this allows processes to access data from network memory or storage servers at close to network bandwidth.

3.1 Basic Mechanisms

The *gms_net* messaging layer includes basic support for typed messages, stub procedures, dispatching to service procedures based on message types, and matching replies with requests. The Trapeze receiver interrupt handler directs incoming messages to *gms_net* by upcalling a registered service routine; the service routine

hands off incoming requests to a server thread. However, *gms_net* it is not a true RPC system: many protocol messages do not produce replies, and there is no support for automatic stub generation. The package is best thought of as a library of procedures and macros used by the messaging stubs to build and decode messages and to direct their flow through the system. It is designed for messages with relatively simple arguments and bulk data payloads (e.g., file blocks) that are not interpreted by the message handlers themselves.

To send a message, a stub allocates a message buffer with *gms_net_makebuf*, calls routines and macros to build the message, e.g., by pushing data items into the message, and sends the message to a destination with *gms_net_sendto*. In Trapeze, *gms_net_makebuf* returns a pointer to a send ring entry, and *gms_net_sendto* releases it. Messages are typed by an operation code and a request/reply bit. Incoming requests are dispatched by using the operation code to index into a vector of registered server-side stubs. Incoming replies are handled directly by the receiver interrupt handler, either by waking up a waiting thread or by calling a reply continuation procedure as described in Section 3.4.

An important function of the RPC layer is to match incoming replies with requests. If a reply is expected, the caller makes an entry in a *call record* table before sending the request message, and places a *reply token* containing a unique call record ID into the outgoing request message. After sending the request, the calling thread or process may block on the call record entry. When the server side generates a reply, it places a copy of the reply token in the reply message. When the reply arrives, the receiver interrupt handler decodes the reply token and retrieves the call record. The call record includes all information needed to process the reply, e.g., by awakening the calling thread or process.

To transfer a page or file block in a request or reply, the stub attaches the memory frame to the message buffer as a payload. The system is inhibited from reusing the frame or overwriting it until the frame contents have been transferred to the network adapter using DMA (Section 2.2). On the receiving side, Trapeze uses DMA to deposit each received payload into a memory frame designated by the receiver.

3.2 Zero-Copy Reply Handling

GMS performance depends on efficient handling of *getpage* replies containing page payloads. When the virtual memory system or file system initiates a page fetch, it first selects the target page frame according to its policies for page replacement and other factors such as page coloring. The goal of the GMS *getpage* client stub is to arrange to transfer the incoming page directly to the waiting frame using DMA.

The RPC system uses the Trapeze incoming payload table (IPT) described in Section 2.3 for this purpose. The client-side *getpage* stub calls Trapeze to allocate an IPT entry and obtain a payload token, which is added to the reply token for the call. When the server-side *getpage* stub generates a reply, it attaches the frame containing the requested page as a payload, extracts the Trapeze payload token, and tags the outgoing reply message by placing the token in its Trapeze header. Back on the client side, the Trapeze firmware recognizes the tag in the message header as the reply payload begins to arrive on the adapter; once the tag is decoded and validated, the firmware initiates DMA of the message payload into the waiting frame.

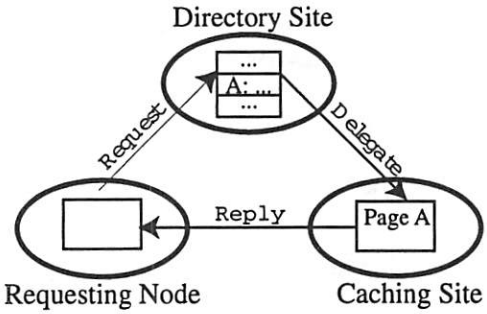


Figure 3: GMS *getpage* operation through a directory site using a delegated RPC.

3.3 Delegated RPC

Unlike traditional RPC, some GMS protocol operations involve more than one server. To fetch a remote page, for example, the *getpage* operation must first locate the page's caching site. To keep track of pages, GMS uses a distributed hash directory for pages potentially sharable by multiple nodes [14]. A requesting node locates a page's directory site by applying a globally replicated hash function to a unique page identifier. It then issues a *getpage* RPC to the directory site, which looks up the page in its portion of the directory, and forwards the request to the caching site. The caching site completes the three-way RPC by returning the page directly to the requester. We call this type of operation a *delegated RPC*.

The key idea behind delegated RPC is to allow the reply token to be passed from node to node until the call is complete; the last node in the RPC sequence then uses the reply token to complete the RPC and reply to the original requester. The delegation is transparent to the requester, which creates its reply token and includes it in the request message exactly as for an ordinary RPC.

With delegated RPC, however, a remote procedure can either return a reply or delegate the request to another peer by generating a new request message with a copy of the original reply token. Each node has the same option of either replying to the original requester or delegating the request to yet another peer. Note that each delegation call is unlike a normal RPC in that the delegated procedure never replies to its immediate caller; it either forwards the request again or replies directly to the node that initiated the delegated RPC. Most importantly, the zero-copy reply handling scheme is preserved, since the Trapeze payload token is embedded within the reply token, and so is available to the node that ultimately generates the reply.

Figure 3 shows how GMS *getpage* uses delegated RPC. The directory site for the requested page delegates the request to the caching site, forwarding the original request parameters and reply token. The caching site generates a reply, attaches the requested page as a payload, and tags the message with the payload token, as described in the previous section. It then sends the reply directly to the original requester, where it is handled in the same way as a direct reply.

3.4 Read-Ahead with Nonblocking RPC

Most file system implementations implement read-ahead to prefetch file data before it is requested. This significantly improves bandwidth for sequential file access, which is easy to detect and exploit [20]. We have extended GMS to support read-ahead in order to meet our goal of delivering files from network memory at full network bandwidth. GMS read-ahead hides fetch latency and delivers peak bandwidth by pipelining the network with a continuous stream of page fetches.

3.4.1 Nonblocking RPC

Any form of prefetching imposes new demands on the communication layer and is highly dependent on its performance. In particular, prefetching requests are RPC calls that generate replies, but the replies must be handled asynchronously and outside of the issuing thread context, so as not to block the issuing thread while the request is pending. NFS client implementations typically solve this problem by handing off read-ahead calls to a system *I/O daemon* that can wait for RPC replies without affecting user processes [22]. This solution requires a context switch to the I/O daemon for each request and response.

To reduce context switching overhead, GMS/Trapeze implements read-ahead and prefetching using *nonblocking RPCs*. To implement nonblocking RPC, *gms_net* supplements the call record with support for *continuation* procedures invoked directly from the receiver interrupt handler to process the reply. These continuations are similar to Draves et. al. [12], but they execute at interrupt time with no associated thread context. Also, each nonblocking RPC call may have several continuations; the issuing stub pushes pointers to these continuation procedures and their arguments onto a *continuation stack* linked to the call record returned by *gms_net_makebuf*. When the reply arrives, the *gms_net* receiver interrupt handler locates the call record for the reply as before, pops the continuations from the stack, and calls them in order with their arguments. Like other interrupt handling code, continuation procedures are not permitted to sleep.

Continuations in *gms_net* nonblocking RPC are related to callbacks in Rover's QRPC [19]. In Rover, asynchronous RPC calls are used to allow applications to tolerate slow and unreliable mobile networks, whereas in GMS/Trapeze their purpose is to support pipelined RPC operations (e.g., prefetching) on a fast and reliable cluster interconnect.

3.4.2 Read-Ahead from Network Memory

GMS/Trapeze activates sequential read-ahead when the file/VM system determines that accesses are sequential, and that subsequent pages are resident in the global cache but not in the local cache. It issues nonblocking RPCs to prefetch the next N pages for some configurable depth N; these requests are issued in the context of the user process accessing the data. Each prefetch request is an ordinary *getpage* operation; to the receiver, they are indistinguishable from synchronous fetch requests. The caller allocates the target page frame before the prefetch, maps it through the IPT as described in Section 3.2, and includes a reply token in the message. The server generates a reply as described in Section 3.2, possibly delegating the request to a peer as described in Section 3.3.

A record of each pending prefetch request is hashed into the local page directory so that the frame can be located if a process references the page before the prefetch completes. If a process references a page with a pending prefetch, the process is put to sleep on a call record until the read-ahead catches up. If no process blocks awaiting completion, nonblocking RPCs do not have specific timeouts. However, call records for nonblocking RPCs are maintained as an LRU cache, so each call record eventually reports failure and is reused if no reply arrives.

When each prefetch reply arrives, Trapeze transfers the payload into the waiting frame and interrupts the host. The interrupt handler uses the reply token to locate the call record, which holds a pointer to the continuation

handler for asynchronous prefetch. The interrupt handler invokes the continuation, which "injects" (hashes) the frame into the local page cache, and enters it into other structures as required, e.g., an LRU list. Note that prefetched pages are not copied.

3.4.3 Deferred Continuations

Since continuations execute from the receiver interrupt handler, they must be synchronized with any kernel code that accesses the same data structures. For example, a file prefetch "inject" continuation could corrupt the internal file/VM data structures if it interrupts a process that was operating on those structures in kernel mode. An obvious solution is to disable receive interrupts for every operation on any data structure that is shared with a continuation procedure, but this would require significant reengineering of existing kernel code that does not expect to be interrupted.

We use an optimistic approach that defers execution of continuations in the rare instances when races occur. Continuation procedures are boolean functions that validate their preconditions by probing the state of relevant kernel locks before executing. If any needed locks are held, this indicates that an operation was in progress when the interrupt was delivered, and the continuation cannot execute. In this case, the continuation returns false with no side effects, and is placed on a *deferred continuations* queue serviced by a kernel daemon thread. Deferred continuations incur higher latency and overhead, but they execute safely. This technique is similar to optimistic active messages [25].

4 Performance

This section presents performance measurements of *gms_net* and sequential file access using the GMS/Trapeze prototype in Digital Unix 4.0. We measure all the RPC variants presented in order to illustrate the costs and benefits of the *gms_net* and Trapeze mechanisms discussed in the previous sections. The file access tests are intended to show the rate at which a GMS/Trapeze client can source and sink data to network storage servers using these communication mechanisms for payload transfer and asynchronous read-ahead. A secondary goal is to show the effect of the operating system kernel interface chosen to read or write file data at these speeds, which are close to the limits of the hardware.

The systems used for these measurements are Alcor and Miata, two DEC Alpha platforms based on the 21164 CPU. Our Alcors (AlphaStation 500 and 600) are clocked at 266 MHz, and use a CIA host-PCI bridge (ASIC pass 2). Miata (PWS 500au) is a newer 500 MHz machine with a Pyxis bridge (ASIC pass 257). All systems are equipped with M2F-PCI32 Myrinet LAN adapters connected through an 8-port switch (M2F-SW8). Both the CIA and Pyxis I/O bridges deliver almost the full bandwidth of the 32-bit 33MHz PCI standard (132 MB/s) in one direction; however, Alcor receives at half-bandwidth and Miata sends at half-bandwidth. Most of the experiments in this section involve a one-way high-volume data transfer: to circumvent the bridge bottlenecks Alcor is always the sender and Miata is always the receiver. Release of the improved Miata-II is imminent, but it is not yet available at the time of this writing.

4.1 RPC Microbenchmarks

Table 1 shows latency and bandwidth results from kernel-kernel RPC microbenchmarks using 16-byte control messages and payload sizes of 0 bytes, 4K bytes, and 8K bytes. In these experiments the request message is a 16-byte control message that generates a reply with an attached payload. The client is a Miata; the server(s) are Alcors. For these experiments, Trapeze was configured to use DMA for control messages in order to reduce overhead at the cost of higher latency.

The table presents measurements for ordinary request/response RPC (2-way) and delegated (3-way) RPC, for three reply-handling variants: traditional blocked caller (*wait*), nonblocking continuation (*cont*), and deferred continuation (*defer*). For the replies carrying payloads, we measured the effect of three payload handling schemes: (1) "solicited" payloads received into frames mapped by the Trapeze IPT, (2) "unsolicited" payloads received into a generic payload buffer attached to the receive ring entry (as for a received GMS *putpage* or *movepage*), and (3) "unsolicited with copy", in which the received payload is copied from a generic payload buffer into a reply buffer not mapped through the IPT. The third variant is intended to demonstrate the value of the IPT for copy-free reply handling.

What is important here is the low incremental cost and high bandwidth of pagesize payloads, and the effects of the payload handling techniques presented in Sections 2 and 3. For example, we can determine from the *2-way wait* numbers that the marginal transfer latency of a solicited payload is about 54μs for 4K and 92μs for 8K, including the cost to map the receiving frame through the IPT. With nonblocking RPCs and continuations, *gms_net* preserves 87% of the 88 MB/s of bandwidth that raw Trapeze provides with 4K payloads on this platform, and over 92% of the 105 MB/s of raw Trapeze bandwidth using 8K payloads. Interestingly, at most 7% of the remaining throughput is sacrificed by copying the payload at the receiver; this reflects the excellent memory system

Msg/Payload		16/0			16/4096		16/8192	
		Latency (μsec)	Bandwidth (msgs/sec)		Latency (μsec)	Bandwidth (MB/sec)	Latency (μsec)	Bandwidth (MB/sec)
2-way	Wait	73.7	13600	Solicited	127.6	32.1	165.9	48.8
				Unsolicited	134.0	30.6	169.6	48.0
				Unsol+Copy	167.4	24.5	217.2	37.5
	Cont	69.0	64500	Solicited	125.8	78.9	163.5	95.1
				Unsolicited	128.0	79.2	166.7	97.4
				Unsol+Copy	160.8	69.7	210.8	90.5
	Defer	73.1	64500	Solicited	131.6	76.9	169.8	94.9
				Unsolicited	135.2	70.7	170.5	96.7
				Unsol+Copy	170.0	50.4	218.3	92.9
3-way	Wait	109.5	9150	Solicited	163.4	25.1	201.9	40.3
				Unsolicited	168.8	24.3	205.9	39.7
				Unsol+Copy	201.8	20.3	254.0	32.2
	Cont	104.6	64600	Solicited	162.6	79.4	199.5	94.7
				Unsolicited	163.6	78.8	202.3	97.1
				Unsol+Copy	195.2	63.6	246.4	91.3
	Defer	108.9	64500	Solicited	166.6	77.6	205.7	95.5
				Unsolicited	169.6	65.7	206.1	96.3
				Unsol+Copy	204.0	47.9	254.4	88.1

Table 1: RPC microbenchmark results for *gms_net* on an AlphaStation/Myrinet network.

bandwidth of the Pyxis-based Miata (this is also apparent on our Intel platforms using the new 440LX chipset).

Several other points are worthy of note. Nonblocking RPCs show a modest improvement in latency because there is no process context switch to handle the reply; however, that benefit is more than lost if the continuation must be deferred. Delegated (3-way) RPCs — which are common for shared file accesses in GMS — exact a high price in latency, but have little effect on bandwidth. Solicited payloads are even cheaper than unsolicited payloads, despite the need to set up and tear down an IPT entry; this is apparently due to the cost of returning the received buffer to the VM page frame pool, and allocating a new one to replace the buffer frame lost from the receive ring entry.

4.2 GMS/Trapeze File Access Speed

We now present the performance of sequential file access on the GMS/Trapeze prototype. In these experiments, the servers are GMS network memory servers with sufficient aggregate memory to hold all the data accessed by the benchmark. Thus all disk access is removed from the critical path, reflecting the "cheating" theme of this paper. The purpose is to view the file system as an extension of the network protocol stack, and measure the bandwidth achievable through the file system interface.

For these experiments, the file system partition where the benchmark files reside is configured to use two variants of the GMS caching policies to improve delivered bandwidth. First, blocks from these files are *sticky* in the global cache: reads of these blocks from network memory are nondestructive, so that each block fetched by a client will occupy memory on both the client and the caching site. This policy uses network memory less efficiently, but duplicated blocks need not be written back to network memory when they are evicted from the client, assuming they are clean. Second, the partition is configured as a *scratch* file system that uses network memory as a *writeback* cache: dirty blocks demoted from local memory to global memory are not immediately written to disk. The writeback policy is unsafe in that file data may not survive failure of a caching site, but it allows file writes to proceed at network speeds, so it serves as a measure of the rate at which a Trapeze client can sink dirty data to a server over Myrinet.

Our results report overhead as well as I/O bandwidth. At Myrinet network speeds, file access overhead is as important as raw I/O bandwidth: it is of limited value to read files at 90 MB/s if overheads consume all of the CPU cycles or memory system bandwidth, leaving the application no resources to process the data. Many of our techniques are targeted at reducing overhead (e.g., by avoiding copies) rather than increasing bandwidth directly.

In fact, there is a complex relationship between overhead and bandwidth. One measure of overhead is system

CPU utilization — the percentage of CPU time spent in the kernel. System CPU utilization grows with I/O bandwidth due to fixed overheads for handling each page of data. For typical applications, user CPU utilization also grows with bandwidth, since the application spends time handling each page as well. As the combined effects of user and system processing push the CPU toward saturation, the user program and the system begin to issue I/O requests more slowly, and bandwidth begins to drop.

4.2.1 File Access Interfaces

Our highest bandwidths and lowest overheads are achieved using the file mapping *mmap* system call, rather than the traditional read/write interface. Differences among these interfaces have a negligible effect on delivered bandwidth at disk speeds, but the effect is substantial at gigabit-per-second network speeds. This is true even on modern platforms whose memory systems have sufficient bandwidth to serve the CPU and the I/O system simultaneously (e.g., Alpha Miata or Intel Pentium-II/440LX). The effect can be dramatic on platforms with lower memory system bandwidth (e.g., Alcor).

The experiments use three different file access schemes. The *stream* option uses *read* and *write* system calls. The *mmap* experiments use a single *mmap* system call to map the entire file into virtual memory, avoiding the copying inherent in the *read* and *write* interface. The *memory-mapped block* (MMB) experiments use a hybrid scheme that combines the benefits of the *stream* and whole-file *mmap* access policies. MMB is an an attractive interface for high-volume file access when performance is important and the uniform addressing of whole-file *mmap* is not required.

MMB uses the *mmap* system call in a block-oriented fashion, repeatedly mapping N regions of virtual memory of size B to different ranges of offsets in the file. Our MMB experiments use $B = 256K$ and $N = 2$ (double buffering). Like whole-file *mmap*, MMB is a zero-copy file access scheme; it incurs slightly higher system call overhead than *mmap*, but our results show that this is insignificant with sufficiently low N and sufficiently large B, and is overshadowed by other benefits. Like *stream*, MMB allows the application to explicitly specify its accesses to the kernel. This information can be exploited by the kernel to improve performance for applications that use MMB. Moreover, MMB is an asynchronous interface, allowing the application to specify accesses early in order to overlap file access with computation.

We have modified our Digital Unix kernels to detect MMB accesses and respond with the following policies:

- When an MMB *mmap* call is issued, the kernel immediately initiates an asynchronous prefetch of all pages in the newly mapped region, superseding the usual read-ahead policy.
- The memory frames that previously backed the remapped region are released, discarding the mapped-over file blocks from the local file cache, and pushing them to the network if necessary. This allows the application to control the local cache replacement policy by selecting the region to remap, and reduces the overhead of the paging daemon and LRU eviction code.
- We have extended the *mmap* interface with a MAP_OVERWRITE flag that allows the application to specify that the mapped file region will be overwritten without reading it; in this case, the kernel may simply leave the existing frames mapped, but change their identity in the file cache. The new flag fixes an inherent flaw in *mmap*: when a process references a page in a mapped region, the kernel does not know if the process will read from that page, so it must initialize the frame by zeroing it or reading it from the file. Our approach will expose corrupt data to a process that mistakenly reads from a region mapped with MAP_OVERWRITE, but it will never violate security by leaking data belonging to another process.

We emphasize that with the exception of the new flag, these policies do not change the *semantics* of the standard *mmap* interface, but only its performance. We leave a detailed study of the MMB interface and its usage to future work.

4.2.2 Sequential Benchmark Results

Figure 4 compares bandwidths delivered to a user process reading and writing files sequentially through the *stream*, *mmap*, and MMB interfaces. To show the effect of user program activity, we report bandwidths and CPU utilizations as the test program touches varying amounts of the data on each page. The benchmark repeatedly accesses a file that overflows the local file cache, varying the size read or written for each page from one word up to the 8K page size. We averaged ten iterations with 120,000 page accesses, moving just under a gigabyte of data to or from the process for each test. Variance is negligible for all tests.

The read tests show that bandwidth starts high and decreases as the test program accesses more of the data. The highest bandwidths are delivered at the left end of the graph; these are sparse read tests in which the application reads and loads only one word of each fetched page. Given the excellent memory system bandwidth on

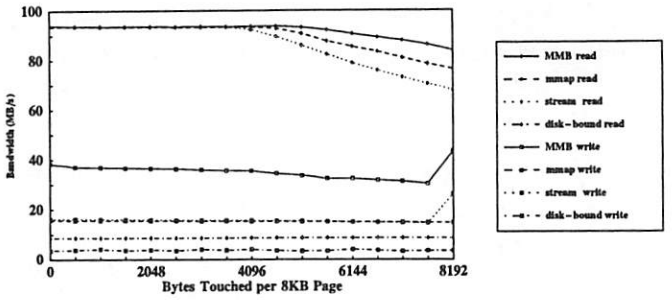


Figure 4: File access bandwidths for sequential read and write tests.

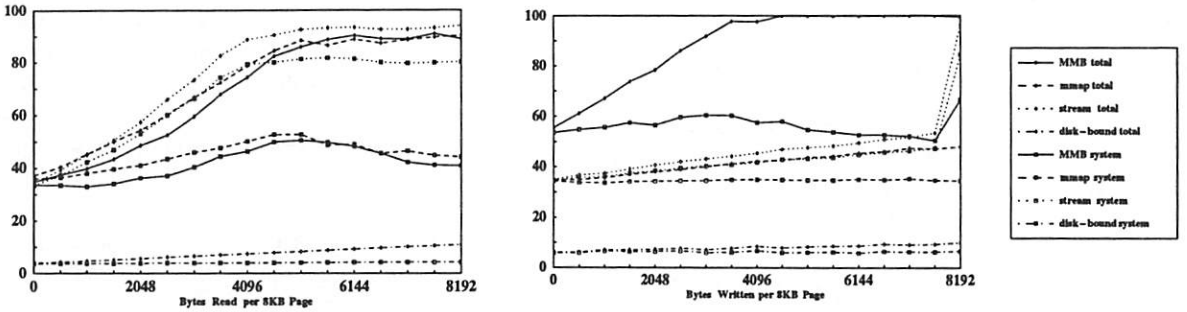


Figure 5: System CPU utilization (%) and total CPU utilization (%) for the read and write tests.

the Miata, the three interfaces deliver the same bandwidth (96 MB/s) until about half the data is touched. Until this point, every access stalls waiting for data to arrive and interface overheads are masked by read-ahead. In comparison, NFS reads from server memory at 13.5 MB/s on the same platform (using standard Myrinet firmware and sufficient I/O daemons); GMS with read-ahead enabled delivers 25 MB/s using IP/Myrinet with the standard firmware rather than Trapeze.

Figure 5 shows CPU utilizations for the same experiments. The system overhead of *stream* rises quickly as the program touches more of its data and the read and write system calls copy more data in and out of the user process. In contrast, *mmap* and MMB avoid the copy, and the system overhead stays relatively flat (the hump at 3-5KB appears to be due to the combined effects of high bandwidth and memory system contention from the user process). In the dense read experiments at the right end of the graphs, bandwidth delivered through *stream* drops to 68 MB/s, as the saturated 500 MHz Alpha CPU spends 80% of its time executing I/O code in the kernel. In contrast, under MMB GMS/Trapeze still delivers almost 84 MB/s, leaving 59% of the CPU time free for the application to process the data. However, simply loading each word up to the CPU saturates the system at these speeds, due to memory system delays. For all three interfaces this experiment is limited by CPU and memory bandwidth rather than the network.

We ran the write tests with Alcor as the client, since its I/O system delivers full send bandwidth. While all tests benefit from zero-copy asynchronous writes (write-behind), file write bandwidths are much lower than read bandwidths for three reasons. First, in the partial-write tests, the kernel must fetch each page (or zero it) before modifying it. Second, these reads do not benefit from read-ahead, since partial sequential writes are rare in practice. Third, Alcor has a slower CPU, its I/O system can receive at only 66 MB/s, and its memory system exacerbates overheads: an Alcor transmitting at full speed delivers less than 25% of its memory system bandwidth to the CPU. Using raw Trapeze, an Alcor can send raw 8KB payloads at 105 MB/s, but the bandwidth drops to 58 MB/s if the sender overwrites each payload buffer before sending it.

For partial writes, MMB delivers the highest bandwidth because it prefetches implicitly on each block access. The bandwidth/overhead spike for the dense write tests at the right end of the graphs occurs because the test program overwrites all of the data, and it is no longer necessary to read each page before writing it. While *stream* and MMB (using the MAP_OVERWRITE flag) recognize this case, *mmap* cannot detect the full-block write in advance, and continues to read before writing. *Stream* delivers 26 MB/s for dense writes on Alcor, while MMB delivers the peak of 46 MB/s (79% of the platform maximum for this test) since it does not copy

the data and also avoids fetching or zeroing the pages before they are overwritten.

5 Conclusion

This paper focuses on features of the Trapeze messaging system that support data-intensive cluster OS services, and their use in the GMS network memory system. The paper makes three contributions:

- It describes useful techniques for copy-free handling of page and block transfers in a network storage system, including an *incoming payload table* (IPT) on the NIC, RPC stub support for zero-copy replies using the IPT, and unified buffering in the network, file, and virtual memory subsystems.
- It shows how to implement useful RPC variants for peer-peer OS services, including delegated RPC and nonblocking RPC using continuations.
- It illustrates use of these techniques in a network memory system that meets aggressive performance goals on a gigabit Myrinet network, using the file system interface to source and sink data at close to network and I/O bus speeds. The GMS/Trapeze prototype features integrated support for high-speed network storage at three levels of the system: (1) the network interface firmware, (2) file/VM and networking subsystems, and (3) the system call interface, with optimizations for the memory-mapped block file access scheme.

6 Acknowledgments

Chandu Thekkath and Geoff Voelker contributed to the original *gms_net* implementation, and some of the ideas in this paper originated with them. Hank Levy and Anna Karlin have contributed to all aspects of the GMS project. We thank Bob Felderman at Myricom and Mark Shand, Don Rice, Marc Viredaz and Lance Berc at Digital for help with the hardware, and Gretta Bartels for help preparing the document.

7 Availability

Trapeze is free software, and a GMS/Trapeze port to FreeBSD is in progress. For information about availability see the Trapeze web site:

`http://www.cs.duke.edu/ari/trapeze`

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K Su. Myrinet - a gigabit-per-second local area network. *IEEE Micro*, February 1995.

[3] José Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 277–291, Seattle, WA, October 1996. USENIX Assoc.

[4] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, October 1996. USENIX Assoc.

[5] Jeffrey S. Chase, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Trapeze messaging API. Technical Report CS-1997-19, Duke University, Department of Computer Science, November 1997.

[6] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[7] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual network transport protocols for Myrinet. In *Hot Interconnects Symposium V*, August 1997.

[8] D. Comer and J. Griffioen. A new design for distributed systems: the remote memory model. In *Proceedings of the 1990 Summer USENIX*, June 1990.

[9] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing (JSPP 97)*, 1997.

[10] Michael D. Dahlin, Randolph Y. Wang, and Thomas E. Anderson. Cooperative caching: Using

remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 267–280, November 1994.

[11] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *Proceedings of IEEE Infocom*, 1997. WUCS-96-12 technical report.

[12] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 122–136, December 1991.

[13] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Hot Interconnects Symposium V*, August 1997.

[14] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[15] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[16] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 1993.

[17] Hsiao-Keng and Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[18] Hervé A. Jamrozik, Michael J. Feeley, Geoffrey M. Voelker, James Evans III, Anna R. Karlin, Henry M. Levy, and Mary K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 258–267, October 1996.

[19] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 156–171, December 1995.

[20] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Unix Operating System*. Addison Wesley, Reading, MA, 1996.

[21] Scott Pakin, Vijay Karamcheti, and Andrew Chien. Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Parallel and Distributed Technology*, 1997.

[22] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.

[23] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[24] Geoff M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98)*, June 1998.

[25] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[26] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects Symposium V*, August 1997.

[27] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.

mhz: *Anatomy of a micro-benchmark*

Carl Staelin
Hewlett-Packard Laboratories

Larry McVoy
BitMover, Inc.

Abstract

Mhz *is a portable ANSI/C program that determines the processor clock speed in a platform independent way. It measures the execution time of several different C expressions and finds the* ***greatest common divisor*** *to determine the duration of a single clock tick.*

Mhz *can be used by anyone who wants or needs to know the processor clock speed. In large installations it is often easier to experimentally determine the clock speed of a given machine than to keep track of each computer. For example, a platform-independent database system optimizer may use the clock speed while calculating the performance tradeoffs of various optimization techniques.*

To run the benchmark long enough for timing to be accurate, mhz *executes each expression in a loop. To minimize the loop overhead the expression is repeated a hundred times. Unfortunately, repetition enables many hardware and compiler optimizations that can have surprising effects on the experimental results. While writing* mhz, *much of the intellectual effort went into the design of expressions that minimize the opportunities for compiler and hardware optimization.*

Mhz *utilizes* lmbench 2.0*'s new timing harness, which manages the benchmarking process. The harness automatically adjusts the benchmark to minimize run time while preserving accuracy, determines the necessary timing duration to get accurate results from the system clock, and measures and accounts for both loop overhead and measurement overhead. It is used throughout* lmbench 2.0 *and can be used to measure the performance of other applications.*

1 Introduction

Mhz is a portable ANSI/C program that determines the processor clock speed in a platform independent fashion, which does not depend on any specific compiler, operating system, or processor. *Mhz* is part of the *lmbench* [1] suite of micro-benchmarks and was used to develop the new timing methodologies for *lmbench 2.0*. *Lmbench*'s guiding philosophy can be described as "accuracy, speed, portability, and simplicity." Each of these tenets impacted the design of *mhz*.

At first glance, determining the processor clock speed seems simple; time the execution of a short number of instructions and divide by the number of instructions. There are several problems with this simple approach, such as the lack of standard clocks with enough resolution to measure the duration of a few instructions accurately. In addition, *mhz* is written in portable ANSI/C that can be compiled into an unknown sequence of instructions of unknown length.

There are a variety of problems that need to be addressed in order to accurately measure time intervals on various processors under various operating systems. On processors with cycle times of 5 nano-seconds, some operating systems have low-resolution clocks, as poor as 10,000,000 nano-seconds, while others have 1,000 nano-second resolution clocks.

Lmbench 2.0 incorporates an entirely new timing harness which automatically controls the experimental system to provide accurate results on all platforms. For example, it determines how long experiments have to run in order for timing results to be accurate within 1% and then controls the experiments so they run just that long. It also automatically corrects for various overheads, such as loop overhead and timing measurement overhead. Considerable effort went into preserving accuracy while minimizing run time, which has paid off in shorter run times (with the same or better accuracy as *lmbench 1.0*) on systems with relatively fine grained clocks.

Determining the clock speed in a platform independent manner is surprisingly difficult because there is no way to measure one clock tick. The inspiration for the solution was based on hazy memories from high school chemistry and physics, of techniques used by nineteenth century chemists and physicists to determine the atomic weight of elements and the charge of an electron [2,3,4].

Section 2 describes *lmbench* 1.0's solution and its limitations, while Section 3 provides some background on computer architecture and how it affects *mhz*. Sections 4 and 5 introduce and describe the newer solution. The experimental methodology used by *lmbench* in general and *mhz* in particular are described in Sections 6 and 7, and the results are presented in Section 8. The Appendix contains a description of the *lmbench 2.0* benchmarking API and a brief tutorial on writing benchmarks using the *lmbench* API.

2 *lmbench 1.0*'s solution

Lmbench 1.0 includes a version of *mhz* that was accurate for a wide range of processors, but contained processor-specific code. It has a single loop, which runs for about a second. The clock speed is the (estimated) number of clock ticks divided by the elapsed time. The number of clock ticks is approximated by multiplying the number of loop iterations (say 10,000) by the number of expressions per loop (1000) and the number of clock ticks per expression. The number of clock ticks per expression is known for some processors and assumed for others.

```
main(int ac, char *av[])
{
    register int a = 1, N = 10000;
    double usecs, mhz;

    start();
    for (i = 0; i < N; ++i) {
        a>>=ac; // expression 1
        a>>=ac; // expression 2
        …
        a>>=ac; // expression 1000
    }
    usecs = stop();
    mhz = N * 1000 / (double)usecs;
}
```

Figure 1

Figure 1 contains pseudo-code for a simple *mhz* that works for many existing processors. It assumes that each shift operation takes a single clock tick. To determine the processor clock speed, just run the benchmark long enough (say 10,000 iterations), and then divide the number of clock ticks (10000 * 1000) by the duration. *Lmbench 1.0*'s *mhz* had processor-specific expressions selected at compile time based on the operating system.

Although the original approach worked on about 90% of the platforms tested, it has several limitations:

- The expressions are processor-dependent.
- The number of clock ticks per expression is not always known a-priori.
- The loop size is fixed and provides no guarantee that the timing interval is significant relative to the system clock resolution.
- The timing loop is only run once, so it is susceptible to errors caused by other independent activity on the processor.

An approach that would be accurate on all modern and anticipated architectures was needed.

2.1 Other approaches

The approach described above requires *mhz* to know the number of clock ticks per expression. This is infeasible since *mhz* is written in ANSI/C and intended to run on a wide variety of processors. We could not find expressions that require a fixed number of clock ticks on all processors. Clearly a method for determining the clock speed that doesn't require such information is needed.

Several techniques were investigated, such as measuring the execution time of two expressions, subtracting the two times, and hopefully getting the duration of a single clock tick. Other techniques include: creating loops with different ratios of two expressions (e.g., a++;a>>=1; and a++;a++;a>>=1 which are 1:1 and 2:1 respectively), and varying the number of times an expression is repeated within the loop. Some of the techniques, such as measuring the difference between two expressions, suffered from the same weakness as the solution in *lmbench 1.0*. Unfortunately, none of these approaches works. At best, most approaches could give the time to execute a single expression, which can already be measured.

3 Computer architecture

Modern computer architectures are complicated and highly optimized. Many of these optimizations are useful for general purpose programs, but can wreak havoc on our micro-benchmark. They make it nearly impossible to predict exactly what happens during execution.

3.1 Superscalar

Super-scalar processors have multiple computational units and can execute multiple operations in a single cycle. Super-scalar processors can also overlap the execution of adjacent instructions, which means the average number of clock ticks per instruction is non-integral [5].

For example, the expression `a+=b+a+a` might be compiled into:

```
ADD   r1,r1,r3  ; r3=a+a
ADD   r2,r1,r4  ; r4=a+b
ADD   r3,r4,r1  ; r1=(a+b) + (a+a)
```

A superscalar processor with two arithmetic units could execute three instructions in two clock cycles by executing the first two instructions in parallel. This would make the average number of cycles per instruction 0.66.

3.2 Instruction reorder buffer

Instruction reorder buffers provide limited workflow-like architecture capabilities to otherwise traditional processors [6,7,8]. The processor keeps track of inter-instruction dependencies and executes an instruction as soon as its data is available (data may be unavailable because it has not arrived from memory yet or because it is the result of an instruction that hasn't completed yet). Unlike dataflow processors, instruction reorder buffers have a bounded (and limited) size, so there is a sliding window of workflow-like capabilities.

Suppose there is a processor with two arithmetic units and one barrel-shifter and the following assembly code:

```
ADD   r1,r2,r3  ; r3=r1+r2
SHR   r3,1,r4   ; r4=r3>>1
ADD   r1,r5,r6  ; r6=r1+r5
```

During execution the CPU will execute the two ADD instructions in parallel because all the arguments are available, and then it will execute the SHR instruction as soon as the first ADD completes.

Instruction reorder buffers combined with super-scalar processors provide the system with a great deal of flexibility and many opportunities for overlapping computations. Unfortunately, that flexibility makes it difficult to craft C expressions that preclude parallel execution.

3.3 VLIW

At least one next generation processor will use a very-long-instruction-word (VLIW) architecture. Each VLIW instruction includes several independent sub-instructions that may execute in parallel [9]. The compiler optimizer technology for VLIW is complex because of this new parallelism.

The next section explains why we see no reason for *mhz* to work incorrectly on VLIW processors.

4 *mhz* solution

Mhz's computes the clock speed using the *greatest common divisor* (GCD) of the execution time for nine expressions, assuming that the execution time for each is an integral multiple of the time taken by a single clock tick. This technique makes no assumptions about the number of clock ticks for any single instruction or the number of instructions used to implement a given expression, except that it executes in an integral number of clock ticks.

To ensure that each expression executes in an integral number of clock ticks (on average), *mhz* uses tightly interlocked operations so processors cannot overlap the execution of the expressions.

Mhz can compute the CPU cycle time if the compiler generates at least two instruction sequences with relatively prime cycle counts. *Mhz* uses several different sequences to increase the chance that two sequences will have relatively prime cycle counts on any given architecture.

The *relatively prime* condition is necessary for the greatest common divisor method work. If all the cycle counts have a common factor (e.g. 2), then the apparent CPU speed will be reduced by that common factor. Also, if there is so much variability in the data that there is no apparent GCD, then *mhz* will return a result that is too large. The instruction sequences are chosen so that there are almost always two sequences with relatively prime lengths.

The processor's clock speed is the GCD of the execution times of the various instruction sequences. For example, suppose *mhz* is trying to compute the clock speed for a 120MHz processor, and there are two instruction sequences:

1.	SHR	(2 cycles)
2.	SHR;ADD	(3 cycles)

If the execution times are:

1.	SHR	11.1ns (2 cycles)
2.	SHR;ADD	16.6ns (3 cycles)

The GCD is 5.55ns and the calculated clock speed is indeed 120MHz. Aside from problems caused by experimental noise, this method should always work with instruction sequences that have relatively prime cycle counts.

Suppose the two instruction sequences have cycle counts that are not relatively prime:

1.	SHR	11.1ns (2 cycles)
2.	SHR;ADD;SUB	22.2ns (4 cycles)

```
double
gcd(double e[], int esize)
{
/* assumption: shortest expression has
 * no more than MAX_COUNT instructions */
#define MAX_COUNT       6
  int   i, j, size;
  double min_e, min_chi2, result, a, b, chi2;
  double *y, *x = (double *)
       malloc(esize*esize*sizeof(double));

  /* find the smallest value */
  min_e = double_min(e, vsize);

  /* {e[j]:j},{|e[j]-e[i]|:i,j,i!=j},{0,0} */
  construct_dataset(e, esize, &y, &size);

  for (i = 1; i < MAX_COUNT; ++i) {
    b = min_e / i; /* clock tick guess */
    for (j = 0; j < size; ++j)
      x[j] = floor(y[j] / b + 0.5);

    /* regression of the samples */
    regression(x, y, size, &a, &b, &chi2);

    if (i == 1 || i*i*chi2 < min_chi2) {
      result = b;
      min_chi2 = chi2;
    }
  }
  free(x);
  free(y);
  return result;
}
```

Figure 2

The GCD will be 11.1ns, and the clock speed will appear to be 60MHz, which is the true speed, 120MHz, divided by the common factor, 2.

Mhz uses nine expressions, which have been carefully designed to minimize this problem. Finding expressions that execute in an integral number of clock ticks on all processors is non-trivial and is addressed below.

4.1 Greatest common divisor

Finding the GCD of the expression execution times can be non-trivial. Since integer arithmetic does not apply to an array of real-valued observations, *mhz* can not do integer arithmetic to find the GCD. In addition, the observations contain noise, which can obscure the true GCD. *Mhz* can, however, compute the GCD by assuming that each C expression executes in an integral number of clock ticks.

Assuming a single clock tick is b nano-seconds, each experimental observation, e_j, can be converted into an integer number of clock ticks, c_j, where c_j = `floor`(e_j / b + *0.5*). The set of points $\{c_j,e_j\}$ should be nearly linear, and the linear regression should have y-intercept 0 and slope b.

Mhz cannot directly calculate b, but it can make a series of educated guesses and choose the best guess. The guesses, b_i, are based on the fact that each experimental time is an integral multiple of b, and are created so $b_i \equiv \min(e_j) / i$. The least-mean-squares linear regression of $\{c_j,e_j\}$ gives a better estimate of b than the initial guess b_i because it is based on all the experimental observations.

The best b_i can be chosen using the chi-squared error of the least-mean-squares linear regression. When $b_i > b$, the chi-squared error will be large because some observations will have poor fits. When $b_i \approx b$ (within the usual experimental error), the chi-squared error will represent the experimental error, and will be far smaller than errors for $b_i > b$. When $b_i < b$ and b is a multiple of b_i, the chi-squared error will be equal to or smaller than the error of b_i because noisy observations may have a slightly improved fit.

Since multiples of the first best fit will have an equal or smaller chi-squared error measure, *mhz* chooses the first fit that significantly reduces the chi-squared error. Comparing the (current) minimum chi-squared error with an i^2 weighted chi-squared error favors previous minimum chi-squared errors and prevents *mhz* from choosing multiples of the correct result.

Figure 2 contains the routine `gcd()`, which computes the GCD. It finds the minimum execution time, `min_e`. `construct_dataset()` creates the dataset y_j, which includes all the experimental measurements e_j, and adds data points with the difference between each pair of observations. To ensure that the regression runs through the origin, it also adds the point (0,0). For each integral number of clock ticks, i, $i \in \{1, 2, \ldots, 6\}$, it computes b_i, the points $\{c_j,y_j\}$, and the least-mean-squares linear regression [10]. The linear regression gives the chi-squared error and a and b such that: $y = a + bx$. If the weighted chi-squared error is less than the minimum chi-squared error, `gcd()` discards the previous result and saves the current minimum.

5 Atomic expressions

Mhz needs simple C expressions that can be strung together without being optimized out of a loop by a smart compiler. The key is to prevent the processor from computing expressions in parallel or overlapping execution of adjacent expressions. Thus each C expression and sub-expression must depend on the result of the previous expression and it must have no sub-expressions that can begin execution before the completion of the previous expression. Otherwise the processor may utilize the inherent parallelism in the expression and overlap the execution of adjacent instantiations of an expression.

This dependency is critical to the design of the C expressions. For example, the expression `a+=a` satisfies the dependency criteria because the next instantiation of the expression cannot be evaluated until the current expression has completed. The expression `a+=b+c` is not completely dependent because the `b+c` sub-expression may be calculated in parallel with the previous instantiation's `a+(b+c)` sub-expression.

5.1 Compiler interactions

Designing the expressions that execute in an integral number of clock ticks (on average) with enough variety to ensure that there are two expressions with relatively prime cycle counts was difficult. The problems were increased by the compiler and processor optimizations and by compiler bugs and limitations.

We experimented with instruction sequences that use pointer accesses to cached memory locations and multi-variable integer arithmetic. Nearly all such expressions are optimized by modern processors that utilize super-scalar processing and instruction rescheduling to overlap execution of adjacent instances of the same expression.

Optimizing compilers gave us a number of headaches because they are able to optimize away many candidate expressions, if they are in simple loops. For example, the expression `a++;` was easily optimized. So we needed to find mathematical expressions that compiler writers either could not or have not bothered to optimize out of a loop.

Sometimes, the optimizer simply discarded the entire loop because the result was not used anywhere. Consequently, *lmbench* is sprinkled with calls to `use_result()`, a dummy procedure whose sole purpose is to fool compilers into thinking its argument is used somewhere else in the program.

Nearly all expressions using several integer variables were useless because they did not interlock correctly, i.e., advanced processors could overlap sub-expressions of the same expression or sub-expressions of adjacent expressions, and consequently, the average number of instructions per expression was non-integral.

There were a few arithmetic expressions that gave one or more compilers trouble (e.g. core dump, infinite loop, or erroneous output):

- a>>=a;
- a+=b+a;
- a+=b;b+=a;

One or more compilers optimized away the following C expressions:

- a+=a; // ADD optimized to a=0
- a&=a; // AND optimized away completely
- a^=a; // XOR optimized to a=0
- a+=b; // ADD optimized to a+=b+b+b+...
- a+=a;a-=a; // ADD;SUB optimized to a=0

The expression `a+=a` can be optimized to `a=0` because our loops contained one hundred copies of `a+=a` in a single iteration of the loop. Each instance of `a+=a` is equivalent to `a<<=1` for unsigned integers. Since C integers have 32 bits (or at most 64 bits), and since one hundred instances of `a+=a` is equivalent to `a<<=100`, the whole loop can be optimized to the single expression `a=0`.

The loop containing the expression `a&=a` can be optimized away because `a&=a` doesn't change the value of `a`. On the other hand, `a^=a` is equivalent to `a=0`, so the loop containing that expression can simply be replaced by the single expression `a=0`. Similarly the sequence `a+=a;a-=a;` is equivalent to `a=0` since `a-a=0`.

The expression `a+=b` provides a wide variety of possible optimizations when put in a loop with a hundred repetitions. One simple optimization is to set `a+=b+b+b+b+...` This allows superscalar hardware to execute multiple sub-expressions in parallel, which means that the number of clock cycles needed to compute the sum is not necessarily a multiple of 100. In addition, compilers may optimize the inner loop to `a+=100*b`.

5.2 mhz expressions

To maximize the possibility that cycle counts will be relatively prime, nine expressions were selected. For example, the expressions `a>>=b` and `a>>=a+a` differ by a single ADD operation, so on most machines their execution will differ by a single clock tick. There are similar small differences between many of the expressions.

The expressions are:

```
1. p=*p;
2. a^=a+a;
3. a^=a+a+a;
4. a>>=b;
5. a>>=a+a;
6. a^=a<<b;
7. a^=a+b;
8. a+=(a+b)&07;
9. a++;a^=1;a<<=1;
```

```
#define MHZ(M, expression)                          \
void                                                \
_mhz_##M (register long n,                          \
          register TYPE **p,                        \
          register TYPE a,                          \
          register TYPE b)                          \
{                                                   \
    for (; n > 0; --n) {                            \
       HUNDRED(expression)                          \
    }                                               \
    use_pointer(p + a + b);                         \
}                                                   \
                                                    \
void                                                \
mhz_##M(int enough)                                 \
{                                                   \
    TYPE        i = 1;                              \
    long        n = 1;                              \
    TYPE       *x=(TYPE *)&x,                       \
    TYPE      **p=(TYPE **)x;                       \
    _mhz_##M(1, p, 1, 1);                           \
    BENCH1(_mhz_##M(n,p,i,i);n=1;, enough)          \
    save_n(100 * get_n());                          \
}
```

Figure 3

Figure 3 shows how the expressions are embedded in the timing harness. Each `MHZ()` macro creates both the function used to measure the execution time of a given expression and the corresponding simple function. Additional pieces of the harness, such as the experimental timing subsystem, are explained below.

Each expression is repeated 100 times in a loop embedded in a simple function (e.g., `_mhz_1()`). Another function (e.g., `mhz_1()`) uses the standard *lmbench* timing macro, `BENCH1()`, to measure the duration of each iteration of the loop in the corresponding simple function. The loop is embedded in a separate subroutine to increase the likelihood that the compilers would utilize register variables as intended.

Different processors can execute the expressions using different instructions and in varying number of clock ticks, but in general there are at least two expressions taking relatively prime number of clock ticks. Also, in each case, the various pieces of each expression are completely dependent and there are no two sub-expressions that can be executed in parallel. Adjacent instantiations of expressions are completely dependent so a processor cannot overlap execution.

6 *lmbench 2.0* timing harness

The single most important element of a good benchmark suite is the quality and reliability of its measurement system. *Lmbench 2.0* includes a timing harness that manages the experimental timing process to produce accurate results in the least possible time. *Lmbench 2.0* gets more accurate results in less time than *lmbench 1.0* by considering clock resolution, auto-sizing the duration of each benchmark, and conducting multiple experiments. Methods for measuring and eliminating several factors that influence the accuracy of timing measurements, such as the system clock resolution, are described below.

The timing harness includes two macros, `BENCH()` and `BENCH1()`, which provide a uniform method for conducting experiments. `BENCH1()` does one experiment and saves the result, while `BENCH()` does eleven experiments using `BENCH1()` and saves the median result. Benchmarked operations must be idempotent so they can be repeated indefinitely.

```
#include "bench.h"
int
main(int argc, char *argv[])
{
        BENCH(lrand48(), 0);
        micro("lrand48()", get_n());
        exit(0);
}
```

Figure 4

Figure 4 shows a complete example of a benchmark that measures the performance of `lrand48()` and reports its performance in micro-seconds. Please see the Appendix for a description of the *lmbench 2.0* benchmarking API and a brief tutorial on writing benchmarks using the API.

6.1 Clock resolution

Lmbench uses `gettimeofday()` to measure the time and compute the time intervals. Unfortunately, `gettimeofday()` has varying resolutions across different flavors of UNIX, and there is no standard method for querying the operating system to find the resolution of the system clock.

Lmbench includes a module, `compute_enough()`, that automatically computes the time interval required to reduce the timing error (due to clock resolution) to less than 1%. The module increases the timing interval until small variations in the measured work produce correspondingly small variations in the measured time. If a 100 milli-second interval is insufficient, the system uses 1second timing interval.

To verify that a timing interval is accurate to within 1%, it determines how many loop iterations consume the desired time, and then jiggles the number of iterations by 0.5% to time the duration of 100.0%, 100.5%, 101.0%, and 101.5% iterations. If the times are 100.0±0.1%, 100.5±0.1%, 101.0±0.1%, and 101.5±0.1%, the timing interval is presumed to be accurate to within 1%.

6.2 Timing overhead

Once the timing interval "enough" has been computed, the overhead of the timing measurements must be measured. The overhead is significant only on systems where the timing interval is relatively short.

The timing overhead is measured by benchmarking `gettimeofday()`. In *lmbench* the timing overhead is the time to exit `gettimeofday()` at the start of the timing interval plus the time to enter `gettimeofday()` at the end of the timing interval, so the time to call `gettimeofday()` represents the timing overhead.

6.3 Loop overhead

Sometimes, the overhead associated with the `for()` loop can be significant compared to the duration of the benchmarked feature, so the loop overhead needs to be measured and subtracted from the execution time. As far as possible, all micro-benchmarks in *lmbench 2.0* have been designed to minimize the impact of loop overhead on experimental results. Micro-benchmarks measuring fast operations have multiple instances of the operation in the loop to reduce the relative magnitude of the loop overhead.

To compute the loop overhead, *lmbench* uses two loops, the first with one instance of an expression and the second with two instances of the expression, giving two equations:

$$T_1 = N_1(loop\_overhead + work)$$
$$T_2 = N_2(loop\_overhead + 2\,work)$$

Where T_1, T_2 are the measured execution times and N_1, N_2 are the loop iteration counts. These equations can be solved for the loop overhead:

$$loop\_overhead = \frac{2T_1}{N_1} - \frac{T_2}{N_2}$$

6.4 Loop auto-sizing

Lmbench 1.0 uses fixed-size loops for many of the benchmarks. The loop sizes were hand-selected to run for about a second on contemporary processors. With processor speeds doubling every eighteen months, *lmbench* needs loops that can automatically scale themselves so the benchmark's accuracy is not compromised by faster processors.

All timing intervals must have the necessary accuracy, but the system does not know a-priori how many iterations are needed to run for the desired time. The experiments are repeated until the experiment runs for at least 95% of the desired time interval. `BENCH1()` adjusts the iteration count after each timing interval. If the measured time is less than 150 microseconds, then the iteration count is multiplied by 10, otherwise the iteration count is scaled by 1.1 times the ratio of the desired time to the measured time.

Some systems with low-resolution clocks return small integral values for intervals smaller than the clock resolution. *Lmbench* assumes that all timing results smaller than 150 microseconds are meaningless and multiplies the iteration count by 10. Otherwise *lmbench* can use the timing information to compute the iteration count needed for the timing interval to be long enough. Since the timing information has experimental noise, *lmbench* sets the iteration count a little larger than necessary.

6.5 Multiple experiments

Lmbench 1.0 reports the results for only one timing interval. As a result, *lmbench 1.0* is vulnerable to independent activity that steals processor time from the benchmark. In practice, the timing intervals are so big that the impact on the results was minimal, unless there is substantial activity. However, *lmbench 2.0*'s shorter timing intervals enabled by the loop auto-sizing and clock resolution detection mean that relatively little independent activity could have a significant impact on a single experiment.

Lmbench 2.0 performs multiple experiments and reports the median result. In general, the median is more robust and stable in the face of noise than the average result [10,11,12].

7 Making *mhz* really work

Mhz has different requirements and sensitivities than the rest of *lmbench*. *Mhz* is more sensitive to small errors in any given experiment than any of the other benchmarks in *lmbench*. As a result, *mhz* includes a variety of techniques to detect or minimize the impact of noisy data on its accuracy. *Mhz* needs the ability to detect when the data is too noisy to generate an accurate result and to detect obviously erroneous data. *Mhz* also needs to be insensitive to single experimental results that are inaccurate.

Since *mhz* measures the clock speed, and since most experimental errors increase the measured time, *mhz* uses the minimum experimental result for each expression, rather than the more standard median.

Mhz determines the experimental results are too noisy to provide a reliable answer by calculating the MHz twice, once using the minimum values for each

expression, and once using the next larger values. If the difference between the two results is less than 1% or 1MHz, then the data is accepted. Otherwise, *mhz* assumes the results are invalid and retries the experiments, or on the third failure, it tells the user the system is too busy.

To reduce the impact of bursts of independent activity on the experimental results, *mhz* does not use the standard `BENCH()` macro. `BENCH()` takes all the measurements for a single expression, so a burst of activity might affect all the timing intervals for a single expression. To spread the experimental error over the all the expressions and maximize the chance of getting some valid results for each expression, the data collection is done in the `main()` procedure in a pair of nested loops. The inner loop iterates over the expressions and the outer loop iterates over the measurements.

`filter_data()` discards results that are obviously outliers. These are usually caused by optimizations that allow the system to optimize a long loop into a few instructions, which makes the number of clock ticks per expression approach zero. Since the C expressions used by *mhz* require a few instructions each, all the experimental results should be within a few multiples of each other. Results further from the median result can therefore be ignored.

The GCD is sensitive to even one noisy value. In order to reduce the impact of any single value, *mhz* computes the GCD for all valid subsets of the data points and chooses the mode (most common value) of the GCDs. Valid subsets have at least two *independent* data points. Data points are *independent* if the execution differs by one or more clock ticks.

Unfortunately, not all data points are *independent*. Some basic C expressions take the same number of clock ticks, but have slightly different experimental times due to noise. The GCD for a set of non-*independent* points will not be a single clock tick. `classes()` ensures that at least two data points appear to have different numbers of clock ticks. Heuristically, they are considered different if the values differ by more than 5%. The subset is ignored if no two points in a subset differ by more than 5%.

8 Results

Mhz has been tested on a wide range of processors, including: PA-RISC (PA-7000, PA-7200, PA-8000), Intel (486, Pentium, PentiumPro, Pentium II), DEC Alpha, PowerPC (PPC-603, PPC-604, PPC-604e), AMD (K5, K6), Sun (MicroSPARC, SuperSPARC, Ultra-I, Ultra-II), MIPS (R4000, R5000, R10000), Cyrix, Cray T3E, and Motorola 68020. It has also been tested on a wide variety of operating systems, including: HP-UX, IRIX, Linux, SunOS, AIX, BeOS, MkLinux, MachTen, OSF1, Unicos/mk, FreeBSD, and Plan9.

We released an alpha version of *mhz* to comp.arch and comp.benchmarks and a cast of volunteers, and received the results for 643 runs of *mhz*. The output of this alpha version of *mhz* includes all the data gathered by *mhz*. Out of 643 runs, 624 runs contained data that would have been accepted by *mhz* as valid. *Mhz* calculated the processor speed within 5% in 611 of 624 runs. *Mhz* had an error greater than 5% in 13 runs. Of those 13 runs, 10 were from one machine that had another CPU-bound process consuming 50% of the processor time, and *mhz*'s result was 50% of the clock speed.

Of the remaining 3 experiments, one was on a Cray T3E running Unicos/mk, and two were on a Sun UltraSPARC II running SunOS 5.5.1. On the Cray, *mhz* reported a clock speed of 633MHz instead of 600MHz. The version of *mhz* used in the experiments included the loop overhead, and the loop overhead measured in this experiment was far too large, artificially depressing the observed times, and inflating the apparent clock speed. We fixed the bug in the loop overhead calculation that caused the problem. On the Sun, the measured times are longer than expected, and the calculated processor speed is lower than expected. We suspect that there were other processes running on the system.

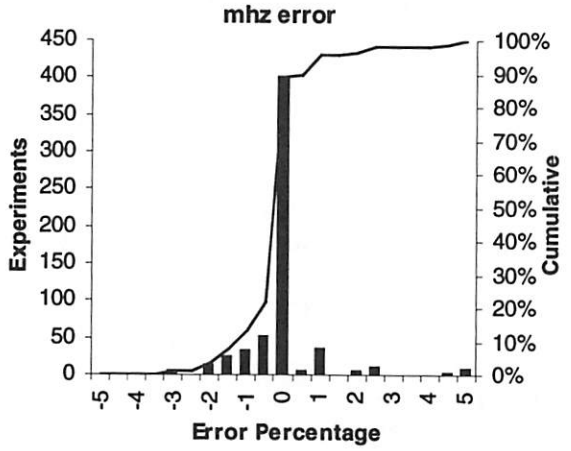


Figure 5

A histogram of *mhz*'s error distribution for the 611 runs is shown in Figure 5. Each bucket represents

0.5% error. *Mhz* is accurate, getting results ±1% 82% of the time, and results ±2% 93% of the time.

Figure 6 includes a selection of results for various processors and operating systems. Please note that the descriptive information is based on information provided by volunteers and may not always be complete.

9 Conclusions

Mhz is a portable C program that can quickly and accurately determine the clock speed of the host processor. *Mhz* demonstrates the utility of a simple mathematical principle: relative primality. *Mhz* also demonstrates many of the experimental and timing features found in *lmbench 2.0*.

Lmbench, including *mhz*, can be downloaded from:

http://www.kernel.org/pub/software/benchmark/lmbench

Lmbench is intended as a toolkit for application and systems programmers to analyze application and system behavior.

10 Acknowledgements

This work would not have been possible without the support, encouragement, and contributions of the many users of *lmbench*. We thank the many people who ran alpha versions of *mhz* and gave us invaluable feedback on its performance and accuracy on a wide range of processors and operating systems. The current program only works because of their efforts.

We would also like to thank Mary Solomon for her research into the experimental and analytic methods of the nineteenth century chemists and physicists, and Prof. Phyllis Brauner and Prof. Len Soltzberg who cheerfully shared their knowledge of nineteenth century chemistry.

Finally, we would like to thank the anonymous referees, Sigal Ar, Fred Douglis, Darryl Grieg, William Long, and Udi Manber for reviewing drafts of the paper, and Patricia Markee for her extensive editorial assistance.

11 Bibliography

[1] Larry McVoy and Carl Staelin, *lmbench: Portable tools for performance analysis*. USENIX technical conference. San Diego, CA. January 1996. pp. 279-284.

Machine	Operating System	MHz
Motorola 68040	4.4BSD-Lite	25
Intel i386	Linux 2.0.33	33
Intel i486	FreeBSD 2.2.2	33
Sun 4m	SunOS 4.1.4	36
DEC R3000	ULTRIX 4.3	40
Sun SPARCstation-10	SunOS 5.5.1	40
Sun SPARCstation-20	SunOS 5.5	50
IBM POWER2	AIX	59
Sun Superserver-6400	SunOS	60
HP 730	HP-UX 10.20	66
Sun SPARCstation-4	SunOS	70
HP 715	HP-UX 9.05	80
Sun SPARCstation-5	SunOS	85
Intel Pentium	Linux 1.2.13	90
Sun SPARCstation-20	SunOS 5.5.1	90
Apple PowerMac 603e	Machten	100
HP 715/100	HP-UX 9.07	100
HP 725/100	HP-UX 9.07	100
Intel Pentium	Plan9	100
Intel Pentium	Linux 2.0.0	100
SGI R4000 IP17	IRIX 5.3	100
Sun SPARCstation-5	SunOS 5.5	110
Cyrix 6x86-P150+	Linux 2.0.33	120
DEC R4400	ULTRIX 4.4	120
HP 770	HP-UX 10.20	120
Sun SPARCstation-20	SunOS 5.5.1	125
Cyrix	Linux 2.1.60	133
DEC AlphaAXP	OSF1 3.2	133
Sun Ultra-1	SunOS 5.6	143
DEC Alpha 347	OSF1 3.0	144
DEC AlphaAXP	OSF1 3.2	150
Intel Pentium	Linux 2.0.0	166
Sun SPARCstation-20	SunOS 5.5.1	166
Sun Ultra-1	SunOS 5.5.1	167
Sun Ultra-2	SunOS 5.6	167
DEC AlphaAXP	OSF1 3.2	175
BeMac PowerPC 604e	BeOS	180
HP 780	HP-UX 10.20	180
SGI R5000 IP32	IRIX 6.3	180
DEC AlphaAXP	OSF1 3.2	190
DEC AlphaAXP	OSF1 3.2	200
HP 899	HP-UX 10.20	200
Intel PentiumPro	SunOS 5.5.1	200
Intel PentiumPro	Linux 2.0.32	200
MIPS 10000	ReliantUNIX-Y	200
SGI R4000 IP22	IRIX 6.2	200
Sun Ultra-1	SunOS 5.6	200
Sun Ultra-2	SunOS 5.5.1	200
IBM PowerPC604e	AIX	233
Sun Ultra-Enterprise	SunOS 5.5.1	248
Sun Ultra-2	SunOS 5.6	296
Cray T3E	Unicos/mk 2.0.2.12	300
IBM PowerPC 604e	AIX	332
Cray T3E-1200	Unicos/mk 2.0.2.12	600
Dec Alpha	Linux 2.0.30	600

Figure 6

[2] R. A. Millikan, *The Isolation of an Ion, a Precision Measurement of its Charge, and the Correction of Stokes's Law*. The Physical Review XXXII(4). April 1911.

[3] Frederick Soddy, *The Interpretation of the Atom*. G. P. Putnam's Sons, New York, New York. 1932.

[4] Arthur Schuster, *The Progress of Physics during 33 years (1875-1908)*. Cambridge University Press, Cambridge, United Kingdom. 1911.

[5] David A. Patterson, John L. Hennesy and David Goldberg, *Computer Architecture: A Quantitative Approach*. Second Edition. Morgan Kaufman. 1996. pp. 221-354.

[6] Anne P. Scott, Kevin P. Burkhart, Ashok Kumar, Richard M. Blumberg, and Gregory L. Ranson, *Four-Way Superscalar PA-RISC Processors*. Hewlett-Packard Journal 48(4). August 1997.

[7] PowerPC 604 Risc Microprocessor, http://www.chips.ibm.com/products/ppc/DataSheets/604/604-180.html. June 1997.

[8] PentiumPro processor dynamic execution, http://pentium.intel.com/procs/ppro/info/dynexec.htm. April 1997.

[9] B. Ramakrishna Rau and Joseph H. A. Fisher, *Instruction-Level Parallel Processing: History, Overview, and Perspective*, Journal of Supercomputing 7(1/2). 1993.

[10] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Second Edition. Cambridge University Press, Cambridge, United Kingdom. 1992. pp. 656-706.

[11] Raj Jain, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, New York, New York. 1991. pp. 183-186.

[12] Russell Langley, Practical Statistics Simply Explained. Dover, New York, New York. 1970. pp. 51-88.

Appendix

The *lmbench 2.0* benchmark API is specified, with descriptions of the timing and reporting functions. In addition, a brief tutorial on constructing benchmarks using *lmbench 2.0* is included. Please see the *lmbench 2.0* release for the complete sources and documentation.

lmbench API

`BENCH(loop_body, enough);`
: This macro is the standard interface to *lmbench 2.0*'s timing subsystem. It repeats the experiment `TRIES` times and reports the median value, unless enough is larger than 100milli-seconds. It uses the macro `BENCH1()` to run each experiment. If `enough` is non-zero, the experiment must run for at least `enough` micro-seconds.

`BENCH1(loop_body, enough);`
: `BENCH1()` is the heart of the benchmarking system. It automatically calculates `enough` and actually benchmarks `loop_body`. It ensures that each experiment is run long enough that the timing errors are minimized. Environment variables `ENOUGH`, `TIMING_O`, and `LOOP_O` affect the initialization of the timing parameters.

`uint64 get_n();`
: Returns the number of times `loop_body` was executed during the timing interval.

`void milli(char *s, uint64 n);`
: Print out the time per operation in milliseconds. `n` is passed as a parameter because each `loop_body` could contain several instantiations of the operation, and there has to be a way to adjust the parameter.

`void micro(char *s, uint64 n);`
: Print out the time per operation in microseconds.

`void nano(char *s, uint64 n);`
: Print out the time per operation in nanoseconds.

`void mb(uint64 bytes);`
: Print out the bandwidth in megabytes per second.

`void kb(uint64 bytes);`
: Print out the bandwidth in kilobytes per second.

Tutorial

Creating benchmarks using the *lmbench 2.0* timing harness is easy. There are two attributes that are most critical for performance, latency and bandwidth, and *lmbench 2.0*'s timing harness makes it easy to measure and report results for both.

The timing harness can be used to quickly create accurate performance measurements for a wide range of features and applications. For example, if a signal processing application needs a fast FFT routine, the programmer could take several implementations, quickly benchmark them, and choose the fastest. Alternatively, the program could include a library of FFT routines and automatically choose the fastest based on the routine's performance on the particular hardware.

There are a number of factors to consider when building benchmarks. The most important thing to

understand is what, exactly, you are trying to measure. If you are trying to find out how long it takes to generate a pseudo-random number, multiply two 500x500 matrices, or copy 1MByte, then *lmbench* can help you accurately measure and report that information quickly. You should also understand the conditions under which you would like to measure the performance. For example, if you want to know how long it takes to copy 4KBytes, then you should understand whether you want to find out how long it takes to copy 4KBytes from and to the cache, or from memory to the cache.

It is useful to form a hypothesis about the feature being measured. Using previously gathered information, it may be possible to accurately predict the performance. Then, build the benchmark, measure the performance, and test the hypothesis. If the hypothesis is wrong, you will have learned something new.

Measuring latency

Latency is usually important for frequently executed short operations, such as memory accesses. Since it is so easy to measure latency using *lmbench 2.0*, it becomes possible to quickly answer questions that arise during system design. For example, simulators may use random numbers frequently, so random number generator performance may be important to overall simulator performance. It takes a few minutes and a few lines of code to measure the performance of a random number generator:

```
1       #include "bench.h"
2       int
3       main(int argc, char *argv[])
4       {
5               putenv("LOOP_O=0.0");
6               BENCH(lrand48(), 0);
7               micro("lrand48()", get_n());
8               exit(0);
9       }
```

Line 1 includes the *lmbench* header, which contains the macros, type definitions, and function declarations for *lmbench*. Line 5 sets the environment variable LOOP_O to 0.0 so *lmbench* won't waste time calculating the negligible loop overhead. Line 6 uses the `BENCH()` macro to benchmark the `lrand48()` function. Since the `BENCH()` parameter enough is 0, *lmbench* will automatically calculate the necessary timing duration. Line 7 uses `micro()` to report `lrand48()`'s performance in micro-seconds.

Measuring bandwidth

The other major component of system performance is bandwidth, which is of primary importance while moving large chunks of data. The mechanics of measuring bandwidth are very similar to those for measuring latency. In many cases the only difference is the function used to report the results.

For example, `bcopy()` is the traditional C-library routine for copying data. It is often heavily optimized because it can measurably affect overall system performance in some commercial benchmarks. A simple benchmark to measure `bcopy()` performance might look like:

```
1       #include "bench.h"
2       #define M       (1024*1024)
3       int
4       main(int argc, char *argv[])
5       {
6               char    *a = malloc(M);
7               char    *b = malloc(M);
8               putenv("LOOP_O=0.0");
9               BENCH(bcopy(a,b,M), 0);
10              mb(M * get_n());
11              exit(0);
12      }
```

Lines 6 and 7 allocate two 1MByte chunks of memory. Line 9 benchmarks `bcopy()`'s performance while copying 1MByte of data from chunk `a` to chunk `b`. Line 10 reports the bandwidth in megabytes per second; during the benchmark timing interval `bcopy()` copied `M*get_n()` bytes.

There are some problems with this particular benchmark because of caching effects when the memory cache is large. Since the `bcopy()` will be repeated several times during benchmarking, the data is more likely to be in the cache, so the benchmark will measure `bcopy()` performance for cached data. If one is trying to measure `bcopy()` performance for non-cached data, this benchmark would need to be modified. For example, allocating larger segments of memory (e.g. 16MBytes) and only copying 1MByte at a time would increase the likelihood of measuring cold-cache results. The modified benchmark would look like:

```
1       #include "bench.h"
2       #define N       16
3       #define M       (1024*1024)
4       int
5       main(int argc, char *argv[])
6       {
7               int     o = 0;
8               char    *a = malloc(M * N);
9               char    *b = malloc(M * N);
10              putenv("LOOP_O=0.0");
11              BENCH(bcopy(a+o,b+o);
12                      o=(o+M)%(N*M);,0);
13              mb(M * get_n());
14              exit(0);
15      }
```

Line 2 defines a new constant, N, which is the number of unique segments the benchmark will use. Line 7 defines the offset variable, o, which is used to select the appropriate segment. Lines 8 and 9 allocate the memory for the segment. The `bcopy()` in line 11

copies data from one segment of a to a segment of b. The offset is updated in line 12 to point to the next segment, modulo the number of segments.

Statistics

It is important to understand the phenomena being measured. Some features, such as clock speed, are constant and variance in the results is caused by experimental noise. Other features, such as context switch times or disk I/O, have intrinsic variance.

The minimum result may be used in place of the median result in some circumstances. For example, when measuring memory latency as a function of actively used memory size, the median result would usually be used because of the variance added by caching effects. However, when determining the cache size, the minimum result might be used.

Special care must be taken when subtracting two measurements, which is usually done when subtracting overhead from an operation. The error in the subtracted result will be larger than the error in the joint measurement since the error in the overhead measurement must be added to the joint error. Also, the subtracted result is smaller so the percentage error has increased. If the overhead is a reasonable fraction of the total measurement, then the error in the result can be significant.

Common pitfalls

Sometimes benchmarks measure effects other than the intended result. There are many ways to make subtle, or even egregious, mistakes in benchmark design. Two common mistakes are measuring partial operations or measuring the wrong operation. Also, think about other factors that can affect the results, such as: caching effects, physical page placement and its effect on direct-mapped caches, process scheduling and cache-stealing in multiprocessors.

Measuring partial operations

Measure the whole operation, not just part of it. If you don't measure the whole operation you can sometimes miss significant features. For example, it is generally accepted that `mmap` is faster than `read` to read a file. If you measure the time to read a file without including the `open`, `close`, and `mmap` overhead, then `mmap` will usually appear to be faster than `read`. However, setting up the mapping is not free and some operating systems, such as Sun's (the origin of `mmap`), have optimized `read` so it is sometimes faster than `mmap`. Unless `mmap` is included in the benchmark, its overhead will not be included in the total cost. When comparing the sequence `open/mmap/bcopy/close` with `open/read/close`, `read` is faster than `mmap` for small files, typically under 32K. In addition, not all operating systems provide read-ahead for `mmap`'ed files, which can result in a 2-3x performance penalty for `mmap` during large sequential read accesses.

Measuring the wrong operation

Measuring the operation you intend to measure. Caches and caching effects are the usual source of problems. It is very important to decide whether you want to measure warm-cache or cold-cache performance. In general it is appropriate and easier to measure warm cache performance. There are other ways to measure the wrong operation, such as allowing overhead to dominate the measurement. For example, *mhz* demonstrates *lmbench*'s ability to measure very short operations, but we limited the loop overhead to a few percent by repeating each operation many times within the measurement loop. Otherwise the loop overhead could have been the dominant feature in the measurement loop.

Multi-process and networking benchmarks are especially prone to errors. Obscure aspects of system design can profoundly impact the benchmark. For example, the behavior of common TCP implementations during connection establishment limited the rate clients generated requests and completely invalidated some common HTTP benchmarks because the HTTP clients only generated requests as fast as the server could service them. The benchmarks never measured server performance during server overload, when the server spends all of its time acknowledging TCP connections.

For more information

There are a variety of sources of information on statistics, benchmarking, and system behavior. [5,10,11,12] are a good starting point for information. The *lmbench* documentation and man pages can help get you started. Also, the *lmbench* micro-benchmarks are a rich set of examples to use when writing your own benchmarks. We hope that *lmbench* will become a standard element of programmers' toolboxes.

Automatic Program Transformation with JOIE

Geoff A. Cohen*
Jeffrey S. Chase
Department of Computer Science
Duke University
{gac,chase}@cs.duke.edu

David L. Kaminsky
Application Development Technology Institute
IBM Research Triangle Park
dlk@us.ibm.com

Abstract

While the availability of platform-independent code on the Internet is increasing, third-party code rarely exhibits all of the features desired by end users. Unfortunately, developers cannot foresee and provide for all possible extensions.

In this paper, we describe load-time transformation, a stage in the program development lifecycle in which classes are modified at load time according to user-supplied directives. This allows the users to select transformations that add new features, customize the implementation of existing features, and apply the changes to all classes in the environment.

The Java Object Instrumentation Environment (JOIE) is a toolkit for constructing transformations of Java classes. An enhanced class loader calls user-supplied *transformers* that specify rules for transforming target classes. We describe some applications of load-time transformation, including extending the Java environment, integrating classes with specialized environments, and adding functionality directly to classes.

1 Introduction

The accelerating use of Java [GJS96] to enable transportable code over the Internet has created both the need and an opportunity to take a larger view of program development. The possibility that programs will consist of components loaded dynamically, possibly from multiple diverse sources, creates challenges for program developers: How can third-party code be adapted to local environments? How do we add functionality to "shrink-wrapped" code? Is it possible to insert new behaviors, such as recoverability, caching, or visualization, into existing implementations?

Fortunately, Java bears within itself the seeds of the solution. Java is an ideal environment for *load-time transformation*, a powerful technique in which user-specified *transformers* (possibly supplied by a third party) add, remove, or change fundamental details of transportable code as it is imported into the local Java Virtual Machine (JVM)[LY97]. Java has several properties that assist load-time transformation. Transportable Java code arrives from the network as compiled *classfiles* containing procedures (methods) and related data definitions for an object type (class): these classfiles retain a great deal of symbolic information, allowing the receiver to determine the structure of the class and to modify it on-the-fly. Secondly, methods are represented as JVM bytecodes: since bytecodes are stack instructions, it is relatively easy to splice new code into existing methods. Finally and most importantly, the JVM uses a user-extensible *class loader* to locate and load new classes on demand: the class loader can be modified to apply load-time transformations to every classfile brought into the local environment.

Load-time transformation has far-reaching implications for the balance of responsibility between class authors and users. In the traditional model, users run programs whose attributes are statically determined by the original authors. Load-time transformation enables a new model, one in which end users assemble and customize applications by chaining together combinations of original code and third-party transformers. The role of the transformers is to implement class features or extensions that the authors did not foresee or chose not to support directly in the original class. Our hypothesis is that many program behaviors are best applied by generic class transformers as needed, rather than hard-wired into the class source. Broadly, transformers are useful for implementing any behavior that is orthogonal to the purpose of the class and can be specified indepen-

*This work is supported by the National Science Foundation under grants CCR-96-24857 and CDA-95-12356. Geoff Cohen is supported in part by an IBM Cooperative Graduate Fellowship. Portions of the work described in this paper were done at IBM Research Triangle Park.

dently. Moreover, support for transformers can improve reuse of existing code by providing a means to adapt it to local needs.

This paper describes load-time transformation using the Java Object Instrumentation Environment, or JOIE[1], a toolkit for specifying transformations for Java classes. JOIE transformers are written in Java and use JOIE primitives to analyze and modify classes. The JOIE toolkit includes an enhanced class loader that invokes transformers at load time. JOIE works with any JVM, and is available for download at `http://www.cs.duke.edu/ari/joie/`.

The remainder of the paper addresses four main questions that arise from the introduction of load-time transformation and its use in JOIE. What are its capabilities? How is it implemented? What are the useful applications? Can it compromise existing guarantees of security and safety?

Section 2 introduces load-time transformation within the context of the program lifecycle. Section 3 presents the environment and the capabilities JOIE offers to enable transformations. Section 4 examines some the implementation details. Section 5 characterizes some broad areas of application of JOIE, and Section 6 presents in more detail the implementation of a specific transformer, Automatic Observable, which adds to classes the ability to detect state changes and report them to registered observers. Section 7 discusses issues arising from load-time transformation, including security, safety, debugging complexity, and legal concerns.

2 Program Transformation

In this section, we take a broad look at the stages of the program lifecycle, and examine in detail the loading stage as a point for transformation. We discuss Java in particular, but the principles apply generally to many other languages and environments.

2.1 Stages of the Program Lifecycle

There are a number of stages in the program lifecycle during which a program author or user can specify the functionality of a class or set of classes. Some examples of tools used at different stages are detailed in Table 1. Originally, of course, the base functionality is declared by the class author in source code, and that source code is translated into an executable by a compiler.

Authors or users can employ post-processors such as instrumentation tools to insert new method calls into an existing executable image. A popular example of this is the tool ATOM [SE94], which works on executable images for the Alpha processor; similar functionality is available for Java with BIT [LZ97]. Most often, this instrumentation is used for performance analysis or as an interface to platform simulation. An important guarantee typically made by instrumentation tools is that the semantics of the original program are not changed. However, Shasta [SG97] processes executable images to run on distributed shared memory systems. Object Design Incorporated's ObjectStore PSE [Obj98] also uses a post-processor, to insert persistence methods into existing code. Rational Software Corporation's tool Purify [Rat98] changes code to detect memory leaks.

Multiple third-party components (classes or more often collections of interacting classes) are integrated during *application composition*. In Java, these components are known as Beans and are often handled in visual builders. This composition allows consumers of code—either end-users or programmers using components in their own application—to modify certain properties of the component. However, users can only modify those properties foreseen by the original author; they cannot independently add features except through the basic object-oriented techniques of inheritence.

After application composition, the classes are eventually loaded into the environment. During execution, the bytecodes can be translated into native local platform instructions by a Just-In-Time compiler (JIT). JITs only reimplement the bytecodes in a different language; they do not add new functionality (although JITs may transform the code for optimization, for example unrolling loops or reordering instructions).

2.2 Load-time Transformation

The architecture of the JVM, in which classes are loaded on demand by a user-extensible class loader, offers a complementary alternative to the previous steps: load-time transformation, in which the loader is responsible not only for locating the class, but for transforming it in ways specified by the user.

Load-time transformation is precisely late enough that the transformation cannot burden other users (as it would if it were performed at, say, component integration), and yet early enough that the JVM is unaware that any transformation has taken place, and the transformed class is still verified by the JVM before it is accepted.

A transformation registered with a class loader can be applied to all classes—or some specific subset—that are eventually loaded into the machine.

[1] Proper pronunciations include *joy*, *joey*, or *zhwa*, depending on your cultural preference.

Stage	Example Use	Example Tool
Pre-processor	macros or conditional compilation	cpp
Compiler	translation from source to classfile	javac
Post-processor	Instrumentation	ATOM, BIT, ObjectStore
Component Integration	Setting text, color	BeanBuilder
Load-time	User-supplied transformation, templates	ClassLoader, JOIE
Just-in-time compilation	Compilation to native code	JIT

Table 1: Stages in the program development life cycle.

2.3 Other Related Work

The idea of load-time transformation itself is not an especially new one: for example, many operating systems and programming environments support dynamic linking, which binds references to library routines at load time. However, these transformations do not alter semantics of the programs themselves (although the semantics of the routines could vary from library to library), and users cannot specify the type or scope of the transformation.

The power of load-time transformation in Java has been recognized by other groups recently. An implementation of parameterized types (i.e. templates) in Java [AFM97] uses the loader to instantiate an appropriate new class. Binary Component Adaptation [KH97] uses load-time transformation to ensure compatibility between third-party components, by performing such symbolic manipulations as renaming methods, marking classes as implementing common interfaces, and restructuring class hierarchies and relationships.

Two other research projects deal with late binding of functionality to objects and classes. "Subject-Oriented Programming" [OHBS94] is a model of object-oriented programming that allows late composition of components from multiple, independently-developed groups of classes. SOP allows new fields and methods to be added, possibly overriding old definitions. "Aspect-Oriented Programming" [KLM+97] emphasizes that different *aspects* of a class (such as the base algorithm, desired precision, data structure, etc.) should be written seperately, to be woven together later into a single class; this simplifies the programming model for modules that contain different aspects.

3 The JOIE Environment

This section outlines how load-time transformation works in the JOIE environment. We first describe how transformers are installed and invoked using the JOIE class loader. We then present the key facilities of the JOIE toolkit that allow transformers to parse, analyze, and modify the transformed classes.

3.1 The JOIE ClassLoader

Before a class can execute in a JVM it must be loaded by a class loader. The JVM invokes a class loader to resolve a reference to a class that has not yet been loaded. The class loader is responsible for locating the missing class file, fetching it from the file system or a network server, and returning it to the JVM. The JVM then *verifies* the new class to ensure that it is semantically valid and safe. To allow for flexibility in loading and instantiating classes, the Java environment allows users to define new class loaders as subclasses of `java.lang.ClassLoader`.

JOIE supports load-time transformation in a special subclass of the ClassLoader. The JOIE ClassLoader exports methods for registering transformers, and it applies registered transformers to each loaded class after fetching the class file into memory but before submitting it to the JVM for verification. The JVM has no way to determine that any changes were made from the original class (although we have adopted the convention of marking transformed classes as implementing an interface `Transformed`, and recording the transformer responsible).

To expose the class internals to the transformers, the JOIE ClassLoader creates a `joie.ClassInfo` object for each class as it is loaded. The transformers access the features of the JOIE toolkit by invoking methods of `ClassInfo` and related classes, as described below. Section 4.1 deals with the internals of the JOIE `ClassInfo` class in more detail.

3.2 Transformers and the ClassLoader

JOIE transformers are written in Java. This design choice offers three important benefits. First, transformers have access to the full power of the Java language, including procedures and variables, conditional logic, and arbitrary control flow. Second, Java vastly simplifies JOIE's design, as no special interpreter for transformers

is needed. Finally, since the transformers are executed by the JVM, they are subject to the same safety checks as any Java program.

A JOIE transformer is simply a Java class implementing the interface `joie.ClassTransformer`. To install a transformer, the user installs a bootstrap wrapper that instantiates the JOIE ClassLoader and the desired transformer classes, and then registers the transformer objects with the ClassLoader. The wrapper then requests the ClassLoader to load the main class of the application to execute; the ClassLoader automatically invokes each registered transformer for each class subsequently loaded by the application. The current version of the JOIE ClassLoader rejects attempts to register new transformers once the first class is loaded. This effectively prevents untrusted application code from installing transformers. This issue is discussed in more detail in Section 7.2.

The core of each transformer is implemented in its `transform(ClassInfo)` method, which is invoked by the ClassLoader to apply the transformation. Multiple transformers can be chained together by passing the `ClassInfo` object for each target class to each transformer in sequence. The ClassLoader invokes the transformers in order of priorities specified at registration time. Within the method `transform`, transformers may call upon the JOIE toolkit routines for reflection, class modification, or bytecode modification, as described in the following subsections.

3.3 Load-time Reflection

A transformer uses the *reflection* portion of the JOIE API to expose the structure of the target class, including symbolic information, fields, methods, interfaces, and attributes. Using the API, a transformer can determine the name, signature, and modifiers (public, synchronized, static, etc.) for any class member. Given a class, it can identify the superclass and any implemented interfaces. Most importantly, a transformer can directly access and browse the class methods and their bytecode instructions.

These reflection features of the JOIE toolkit are similar to those present in a number of languages to allow the discovery of the structure of classes at run time. Java added runtime reflection functionality in the 1.1 release of the Java Developer's Kit (JDK). However, the Java reflection API is available only after the class has been loaded into the JVM. This is too late for load-time transformation, which seeks to transform the class before loading takes place. Additionally, reflection was not designed to extend functionality, and so does not make available the implementation of class methods. Method implementations are accessible through the `javap` disassembler included in the standard Java Developer's Kit (JDK), but `javap` runs from the shell and prints to its standard output; it is not integrated into the Java reflection API, nor does it produce a data structure that can be manipulated by the program.

JOIE's reflection features are useful for a wide variety of analyses, both in their own right as program analysis tools and as the basis for more powerful transformations. For example, JOIE can be used to construct the class hierarchy, build a call graph, or create interprocedural control-flow or dataflow graphs.

3.4 Class Modification

Transformers call upon the JOIE API not only to discover elements of classes through reflection, but also to change or modify those elements. That is, given access to a class (`ClassInfo`) or a member of a class, a transformer can change aspects of the class implementation. For example, transformers can: set or unset modifiers; add, remove, or rename fields or methods; change method signatures or field types; adjust the list of interfaces implemented by the class; adjust references to fields or methods to point to new fields or methods; adjust the value of embedded constants; or manipulate the inheritance hierarchy.

The structure of Java classfiles allows the JOIE toolkit to implement a wide range of class modifications without the need to modify other classes that refer to the target class. This is because all references to class members, including external references, are symbolic and stored in a *Constant Pool*. Section 4.1 discusses implementation issues for class modification in JOIE.

3.5 Bytecode Modification

The power of JOIE stems primarily from its ability to modify the bytecode instructions as well as the fields and properties of a class. Given a `ClassInfo`, the JOIE reflection API allows the transformer to iterate through each `Method` of the class. Given a `Method`, the API exports an array of `Instruction` objects along with ancillary information such as parameters to the method, maximum stack and frame size, exception handling information, etc. The transformer can reorder or replace instructions, insert new instructions, alter the stack depth, reassign values to different frame locations, or modify exception handling by inserting new handlers or modifying the range of instructions protected by a handler. One important use of these features is to *splice* new code into the existing implementation, changing the behavior of the method without affecting the code that is already there.

Class	Method	Explanation
ClassInfo	getMethods	returns array of Methods
ClassInfo	getFields	returns array of Fields
ClassInfo	addField	inserts new Field in class
Field	set	sets a flag (such as private)
Method	getCode	returns Code for Method
Code	getInstructions	returns array of Instructions
Code	getExceptionTable	returns exception-handler table
Code	insert	inserts new Instructions into Code
Instruction	getMnemonic	returns string mnemonic
Instruction	getOp	returns first operand of instruction

Table 2: Selected methods from JOIE API.

Transformers can use JOIE's bytecode modification features for a variety of purposes. For example, transformers may instrument classes by inserting static method calls to global analysis routines, or instructions to increment per-instance counters. Another use is to add exception handling code to allow classes to run in an environment that can produce exceptions not anticipated or handled by the original class author. Such a transformer would contain a list of unsafe operations together with a body of code to deal with exception types that could be raised by those operations. The transformer installs the code as an exception handler in each transformed class, and updates the classfile's table of exception handlers to reflect which regions of code are protected by that handler.

Splicing can be combined with class modification to enable a powerful new way to extend functionality. The transformer can add new generic methods that implement some functionality within the class, then splice in calls to those methods at appropriate points. For example, a transformer that adds persistence methods for moving objects to and from a persistent store could also splice calls to those methods in constructors and finalizers. Section 5 outlines potential applications in more detail, and Section 6 presents a case study of such a transformation.

4 Implementation of the JOIE Toolkit

This section outlines the implementation of the JOIE toolkit in more detail, and addresses several implementation issues for transformation in the Java environment.

4.1 ClassInfo

JOIE transformers operate on `ClassInfo` objects, which encapsulate and export all of the information present in class files. The JOIE ClassLoader creates a `ClassInfo` object for the class before invoking the first transformer, then passes the `ClassInfo` to each registered transformer in sequence. After the last transformer completes, the ClassLoader converts the `ClassInfo` into a transformed classfile byte array in memory, and submits it to the JVM for verification.

To improve performance, the JOIE toolkit methods parse the elements of the target classfile lazily as they are requested by transformers. For example, if a transformer simply adds a new interface to the list of interfaces implemented by the class, then `ClassInfo` need not parse the list of fields, methods, or bytecodes. Since each `ClassInfo` instance preserves parsing state across transformers, each element of a transformed class is parsed at most once.

Java classfiles retain a great deal of symbolic information in a data structure known as the *Constant Pool*, including variable-length symbol entries for all methods, fields, and constants defined or referenced by the class. Entries in the Constant Pool may address other entries by index. For example, an entry for a method `Banana.peel()` contains the index of the entry for class `Banana` and the index of an entry holding the name and type of the method. The entry for class `Banana` in turn contains the index for the UTF-8-encoded string "Banana." The name and type are also stored as indices that point to encoded strings for the name ("peel" in our example) and the descriptor, which encodes parameter and return types.

All references in instructions — including field accesses and method invocations — are represented as indices pointing to symbolic names stored in the Constant Pool. The JVM resolves external references by symbolic name as the program executes.

This structure allows the JOIE toolkit to implement a wide range of load-time class modifications easily. In particular, it is rarely necessary to update classes that reference the transformed class, since all references into the

target class are resolved by symbolic lookup in their own Constant Pool. Moreover, many class modifications can be implemented simply by adding or manipulating individual entries in the Constant Pool. In particular, it is unnecessary to modify the bytecode instructions in the target class unless the transformer explicitly requests it. A change to the Constant Pool affects method instructions only if it changes the index of an entry in the Constant Pool. Fortunately, there are no ordering constraints on the Constant Pool, so any new entries can be appended rather than inserted in the middle, preserving the indices of existing entries. Unreferenced entries are simply left in place.

As the JOIE toolkit parses the Constant Pool, it creates a list for each type of entry, storing each symbol value and its index. This allows us to more quickly add new entries without duplication. For example, to insert a new entry for a Field we first search the list of field entries to ensure that an entry with the requested class, name, and type is not already present. If not, we construct a new field entry, and must then search for or create appropriate class, name, and type entries.

4.2 Splicing and Inserting Bytecodes

In a typical JVM, each method invocation runs with an operand stack and a single index-addressable *frame* whose size is statically determined. JVM bytecodes operate on elements at the top of the operand stack or move operands from the frame to the operand stack or vice versa. For example, the `iadd` instruction pops the top two elements off the stack as integers, adds them, and pushes the result onto the stack. Similarly, a method invocation pops the target of the method and the appropriate number of parameters, and places the result (if any) onto the stack.

This architecture has interesting implications for bytecode modification. First, being stack instructions, bytecodes are sensitive to placement and ordering. For example, simply inserting a method call instruction into a sequence might consume the wrong value off the stack, and leave an unexpected value on the top. In general, splices must be *stack-neutral*, i.e., the spliced code must leave the depth and types of the stack unchanged. However, the values of the entries in the stack may be changed, and the splice may have other side effects that affect the rest of the method, such as modifying the frame or some object.

A second issue is that all branches are relative, so inserting instructions between a branch and its destination will make the destination field incorrect. To preserve the original control flow, JOIE must correctly update the destination field. JOIE solves this by applying modifications to collections of Instruction objects linked to the `ClassInfo`. The Instruction objects represent branch targets as pointers to the destination Instruction. A similar approach is used to update the exception handler table, to preserve the binding of exception handlers to ranges of instructions. In each case, JOIE regenerates the relative addresses when it linearizes the `ClassInfo` to a classfile. For load-time transformation, the JOIE ClassLoader generates the transformed in-memory classfile with the correct relative offsets before submitting it to the JVM for verification.

4.3 Modifying the Frame

As described above, each method invocation places the method's arguments and local variables in the frame. Thus any method transformers that change the number of arguments or local state of the method must modify the method's frame.

Modifying the frame is somewhat trickier than modifying the Constant Pool. In particular, the JVM specification requires that the arguments to the method appear in order at the low end of the frame, before local variables held in the frame. This means that adding a new argument to a method may displace existing local variables or even other arguments. In these cases, the transformer must update the method's instructions to redirect loads and stores from one frame location to another.

Relocating references to frame locations is simple in concept but more complex in implementation. There are twenty-five separate instruction types responsible for loading operands from the frame to the stack, and twenty-five complementary instructions for storing from the stack to the frame. Since the instruction set is strongly typed (for ease of verification), each type (integer, long, float, double, and reference) has its own family of loads and stores. Moreover, JVM load and store instructions encode their operands in any of several different ways. For example, references to frame locations 0, 1, 2, and 3 may use efficient "shortcut" instructions with the location encoded in the opcode. If the frame has more than 256 locations, load and store instructions that reference locations at the high end of the frame must be prefixed by a "wide" bytecode, indicating that the frame offset is specified as two bytes rather than one. In these cases, changing the frame offset of the operand may force a change to the opcode itself, or make it necessary to insert or delete a "wide" prefix. For example, the instruction could be a shortcut that references frame location 3; if the new target is location 5 then the opcode must be changed to take an explicit operand, since there is no shortcut for frame location 5. JOIE correctly generates the correct opcode and operand in the face of such changes.

4.4 Performance Issues

Delays for loading classes are already substantial, given the high latency of loading the class from disk or from across the network, and the time required to verify the class. Even so, the performance of load-time transformation is a critical concern, since the overhead of transformation may be perceived by the user.

Despite the incremental parsing optimization, the class parsing and modification code in our current prototype is fairly slow. While we have put little effort into performance tuning at this point, initial experiments with simple JOIE transformers show that it takes milliseconds or even tens of milliseconds to parse and transform typical classes. We did these experiments on 167 MHz Sun Microsystems Ultra 1 systems with Solaris 2.4, and using Sun's JDK 1.2 beta 3 and JIT compiler.

We plan to improve our current JOIE prototype in several important ways. `ClassInfo` currently represents all internal data structures (lists of methods, fields, interfaces, and instructions) as Java arrays. This makes access fast (as compared to Vectors, for example), but it also makes insertion extremely expensive. In a future release we plan to instead use the Java Collection interface available in JDK 1.2. In addition to improving performance, this will allow transformers to use the iterators and update primitives in the Collection interface to access the Methods, Fields, and Instructions of each target class, streamlining the JOIE interface and implementation.

Adding fields or methods to a class often requires repeated searches of the Constant Pool, since we are forbidden from repeating any entry. Currently these searches are linear; an index structure would significantly reduce the cost of adding and accessing entries in the Constant Pool during transformation.

5 Example Applications

There is a vast array of different types of applications of JOIE. In this section we consider three broad categories: extending Java, integrating classes with a system infrastructure, and adding functionality to classes. These examples are meant to be suggestive, not exhaustive. Other uses of JOIE include instrumentation, program analysis, and optimization.

5.1 Extending Java

There is always a demand for new features in Java, but it is difficult to extend the environment while preserving Java's "Write Once, Run Anywhere" promise. New features for Java could be implemented at a number of different levels: the source language, the compiler, the JVM, or the JIT compiler. JOIE adds a new level where extensions to the environment can be prototyped easily and safely. Once the extensions prove useful, they may be implemented at other levels, or perhaps remain implemented as a JOIE transformer. More generally, transformers can be used to run non-conforming code on standard JVMs, or conforming code on non-standard JVMs.

For example, consider the demand for support for complex numbers as primitives. While a developer could add such a feature to a compiler and matching JVM, classfiles produced by the compiler would not run correctly on any standard JVM. Alternatively, one could use a Complex class, with appropriate methods for adding, multiplying, etc. This solution is portable but inefficient.

However, a transformer could recognize the instantiation of Complex, and replace each instance with a pair of floats, and similarly replace method invocations with inline floating point operations. The transformed code will incur lower overhead than the relatively expensive method invocation, while conforming to the JVM standard. More importantly, the classfile running in a system without JOIE will still run correctly, if more slowly. Adding such a feature to Java does not require the transformer author to rewrite or even look at JVM source code.

A different style of extension could add new bytecodes to the JVM implementation. For example, the performance of numerical analysis programs may be improved by augmenting the instruction set with vector bytecodes implementing matrix operations such as sum, dot product, min, max, etc. A compiler could easily generate code using the new bytecodes, but the code would not run on any JVM that does not implement them. Assuming that a transformer could recognize these constructs, it could replace the standard long version of the code with the new bytecodes (or vice versa).

Parametric types are another example. The Thor group proposes an implementation of parametric types for Java [MBL97] that requires a modified JVM and compiler. A load-time transformer could translate a less efficient but standard implementation to use special new instructions.

More ambitious language extensions could add pre- and postconditions to methods [Mey92], continuations, closures, or multimethods. An implementation of "security-style passing" at Princeton [WF98] uses JOIE to achieve a feature similar to continuations, making some subset of the state of the computation available at all times.

5.2 Integrating Classes with a System Infrastructure

Other transformations change classes to integrate them into some runtime environment. This idea generalizes the use of ATOM or BIT to instrument code to drive an on-the-fly simulation, by inserting code to report selected events (e.g., branches or method calls) to the simulator. This technique can be used for other purposes as well. Some possibilities include integrating imported classes with local system environments that support object caching, distribution, debugging, or visualization.

The transformers for these environments have similar structures. First, the transformer must identify the points in the code that handle events of interest to the system environment, e.g., object instantiation or method invocation. The transformer then splices calls to the environment at these points, passing any necessary information as arguments.

For example, a debugging environment may be interested in the values of parameters to methods, return values, etc. A debug transformer would identify areas of interest and insert calls to the environment, allowing a user perhaps to visualize and see results of the computation. The advantage of this approach is that the debugging calls are inserted only at load time; if the user does not require them, they are not inserted at all, reducing the size of the transported class. Also, note that running the debug version of the code requires no special JVM, and only areas of interest are transformed.

5.3 Adding Functionality to Classes

More ambitious transformers seek to extend the functionality of classes themselves as they are loaded. While any sort of functionality could be added in principle, this approach makes most sense when the new functionality is orthogonal to the implementation of the class, such as persistence or logging. This makes the transformer easier to construct, and also creates a logical separation between the code of the original class and the new functionality. Users who are uninterested in that specific function avoid the cost of downloading, verifying, and storing the unwanted code.

An existing approach to the late extension of classes is known as *mixins*, which are implemented in LISP [MSW83] and other languages. Mixins are groups of methods and fields that are added to a class definition sometime after compilation. Traditional mixins are passive: they add methods to the class, but those methods are called only from outside of the class, and never by the original methods.

JOIE can be used to implement a stronger model, *active* mixins. An active mixin is structurally similar to a traditional mixin. In addition to adding fields, methods, or interfaces to an existing class, active mixins parse existing methods and insert or replace instructions. For example, a mixin for versioning would contain methods for reading the appropriate version, and writing changed instances of objects to a storage system. However, these methods must be explicitly called by external code. An active mixin for versioning would not only add the methods, but also insert a read call in the constructor, and a write call in the finalizer.

Recoverability could also be implemented as an active mixin. Methods could be made recoverable by inserting logging code: whenever these methods finish, parameters or return values are written to disk, by calling methods supplied by the mixin author (possibly inserted as methods into the class, or an auxiliary class). With the addition of a mixed-in method to read the log and begin computation at the point of failure, simple recoverability could be added to a class.

6 Case Study: Automatic Observables

In this section, we describe an active mixin called Automatic Observable, and discuss its implementation in JOIE.

6.1 Observer/Observable

The Observer pattern is a well-known design pattern (also known as Model/View), discussed in [GHJV95]. In it, whenever an *Observable* object changes state, its *Observer* classes are notified. This pattern is used when there are common classes that encapsulate state, and other classes that represent a "view" of that state.

Sun's JDK provides an implementation of Observer/Observable in Java. In it, Observer is an interface containing the single method `Update(Observable, Object)`, which is called by an Observable object's `notifyObservers` method whenever it wishes to inform its Observers that its state has changed. User classes acting as Observers need only implement the interface, call the `addObserver` method of an Observable instance, and they will be able to begin observation.

In contrast, Observable is implemented as a class which includes methods for managing Observer lists and notifying Observers, as well as code to check, set, and clear a hasChanged bit. Programmers define Observable classes by subclassing the `Observable` base class. Unfortunately, requiring authors to use a specific base class in this way is often limiting. For example, third-party software cannot be made Observable without changing an existing class hierarchy.

```
class AutoObservable implements ClassTransformer {
    ClassInfo transform(ClassInfo cinfo) {
        cinfo.addField(Type.BOOLEAN, "dirty");
        Instruction[] splice = code.putField("dirty", true);
        Method[] meths = cinfo.getMethods();
        for(int i=0; i<meths.length; i++) {
            if(!meths[i].isSet(Modifier.STATIC)) {
                Code code = meths[i].getCode();
                Instruction[] insts = code.getInstructions();
                for(int j=0; j<insts.length; j++) {
                    if(insts[j].getMnemonic().equals("putfield")) {
                        code.insert(j+1, splice);
                        insts = code.getInstructions();
                        j += splice.length; }   }   }   }   }   }
```

Figure 1: Code Fragment From JOIE Transformer for Automatic Observable

6.2 Observable Mixins

The problems with the Observable base class could be avoided through the use of mixins. An Observable mixin would mix the Observable code, including the hasChanged bit, into an existing class, without modifying the inheritance hierarchy. However, with this approach, the class author is responsible for properly instrumenting the code to manage the hasChanged bit and call notifyObservers at appropriate times. Unfortunately, the traditional mixin model is not capable of changing existing methods to include new method calls.

However, Observable is easily implemented as an active mixin using a JOIE transformer. Figure 1 shows a fragment of the code for the JOIE transformer for Observable. The implementation resides in the single method `transform` called by the JOIE ClassLoader (of course, transformers may have additional methods). The transformer first adds a new boolean field, "dirty", to the class. It then scans each instruction of each non-static method defined in that class. If the instruction changes an instance field (e.g., the `putfield` instruction), the transformer inserts a code sequence that sets the dirty bit. The counter skips over those new instructions, and the scan continues. Not shown in the example is code that inserts the method call to notify the observers, or to mix in the methods to return the dirty bit, notify observers, or manipulate the list of observers.

This transformer is meant to be illustrative only. A more efficient scheme would build and traverse the control flow graphs, setting the dirty bit at most once per path, and adding the notify call only to exit blocks that could follow a block containing a set to dirty. A more universal solution would also perform an interprocedural search on member objects, and also handle inheritance (only a superclass needs to have the methods and the dirty bit mixed in, but all subclass methods must be scanned and spliced).

7 The Key to Pandora's Box?

Introducing a new phase to specify functionality raises a number of difficult issues. These are generally a result of the fact that traditional tools, from design tools to debuggers and security systems, were not designed to take load-time transformation into account.

We suggest that load-time transformation will be a permanent part of the programming model in the future. However, much work remains to be done to develop a safe, general, and practical methodology for load-time transformation. Here, we touch on four important issues: debugging, security, legal issues, and the need for a framework to guide transformer authors in writing transformers that are general and correct as well as powerful.

7.1 Debugging Transformed Code

When a bug in a transformed application arises, it is often unclear if it is a bug in the original code, a bug in the transformation code, or a bug that arose from the interaction of the two. Debugging JOIE transformers today demands an understanding of the both the original code and the transformer. The problem is particular severe if multiple transformations were active. JOIE currently has no support for controlling interactions among transformers.

Some specific debugging support is essential to facilitate load-time transformation as a useful programming tool. At the very least, the environment must be able to

determine if the failed code was transformed, if transformed code appears in the call chain, and which transformers were responsible for any transformed code. Basic information can be made available to the environment by tagging transformed classfiles in JOIE.

7.2 Security

A natural reaction when presented with JOIE is to worry that it will break the security guarantees made by Java, e.g., the promise that imported code cannot gain unauthorized access to resources or memory. With respect to security, JOIE transformers are no more powerful than any imported code. Put another way, code produced by JOIE is as safe as code produced by an untrusted compiler. However, it is important to develop a rigorous understanding of the effect of transformers on security. For example, while class authors choose their compilers, they do not necessarily choose their transformers; a transformed digitally signed applet may not be as trustworthy as the original.

Java security is supported by two main pillars: the verifier and the SecurityManager. The Java verifier guarantees, among other things, that code is type-safe, and particularly that pointers cannot be created or manipulated. This prevents a malicious or buggy program from walking through memory and reading or writing where it normally would not have access.

The SecurityManager is a user-extensible class that is asked by library code at execution time whether specific classes have permission to access specific resources. For example, a program loaded over the Internet might not be permitted to read from the filesystem, but it could spawn a new thread.

Since transformations are applied before the code is verified, transformed code must still comply with the same type-safety restrictions as non-transformed code. The SecurityManager has the same ability to restrict transformed code's access to system resources as before.

There are also important security policy issues. For example, is it safe for classes to instantiate and register new transformations during the execution of the program? Can transformers be loaded and applied from over the network safely? Currently, JOIE's default policy is to disable registration of transformers as soon the first class is loaded, but this can be overridden by the user. There is also the question of how to determine the access permissions of a transformed class: can classes ever gain or lose permissions due to being transformed?

7.3 Legal Concerns

An additional concern covers legal issues, which are beyond the scope of this paper and the authors' experience. One concern is that JOIE, like many utilities, including decompilers, disk copy routines, and binary editors, could be used to violate copyrights. However (like those other tools), it has legitimate uses that remain within the law.

A more complex and troublesome issue regards liability: is a failure the result of the original class, the transformer, or both? Who has the ultimate responsibility?

7.4 Usability

To be useful, the software should also be convenient to use. Currently, transformer authors must be conversant with the bytecode structure of Java and have a deep understanding of classfile structure and layout. A set of abstractions that sit on top of JOIE might provide a framework in which less experienced programmers can construct useful and correct transformations. We are investigating the development of such a framework.

8 Conclusion

This paper describes JOIE, a toolkit for automatic, user-directed transformation of Java classfiles. Program transformation using JOIE can be the basis for a wide variety of extensions, including instrumentation, mixins, automated inclusion of code for visualization or debugging, and bytecode replacement.

A powerful feature of JOIE is its ability to apply transformations at load time to meet the user's site-specific needs. Load-time transformations can include changing the implementation of the original methods of the class, introducing a new model for extending the functionality of Java classes.

Load-time transformation is well-suited to use with transportable code in the Internet. It allows software consumers to assemble customized applications by combining software modules and transformers obtained from a variety of sources. To illustrate the potential of load-time transformation, we outline several examples of how it can be used to extend the Java programming environment, integrate classes with system facilities in the local environment, and adapt or extend third-party classes. To show how these transformations would work, we described in detail the Automatic Observable transformer, an *active mixin* that adapts existing classes to use the Observer/Observable design pattern.

The introduction of program transformation — and load-time transformation in particular — raises many interesting research issues and practical concerns that must be addressed. A fundamental question is: when is it appropriate to specify functionality as a transformation? Other issues include the problems of debugging

transformations and transformed code, security and legal concerns, and performance. Most of these issues are common to any introduction of a new stage in the program development lifecycle. This paper does not attempt to present complete answers to these questions. Rather, the contributions of the paper are to show the potential of load-time transformation, present a prototype toolkit as a basis for further experimentation, and identify the key issues raised by the technique.

There are a number of features that we plan to add to JOIE, including more advanced dataflow analysis for methods. We are also extending a Java compiler to pass programmer-supplied hints to JOIE. Such hints would be useful, for example, to indicate blocks of code that could be transformed to superoperators[Pro95], such as dot product or a method invocation on a parametric type. Also, JOIE adds functionality at the class level only. We are considering extending this idea to instances, allowing orthogonal features to be added on an instance-by-instance basis.

Our plans for future work with JOIE include exploring new uses for load-time transformations. JOIE is most exciting when viewed as an enabling technology. We believe that JOIE allows a fundamentally more powerful way to express functionality, and intend on exploring the possibilities. The Automatic Observable active mixin suggests that transformers can automate other design patterns [GHJV95] as well. Other promising possibilities include automatic hooks and adaptation for system facilities such as persistence, distribution, recoverability, survivability, caching, and mobility.

9 Acknowledgements

We'd like to thank Mike Fox, Jim Gray, Manoj Kasichainula, R. Adam King, and Zhiyong Li of IBM Research Triangle Park for assistance and guidance from the earliest stages of the project. Thanks to early users Jonathan Seeber and Anya Bilska for putting up with bugs. Siddhartha Chatterjee and Jan Prins of UNC-Chapel Hill, and Owen Astrachan and Robert Duvall of Duke University all made very useful comments on drafts of this paper. Chris Laffra and Lee Nackman of IBM's Thomas J. Watson Research Center gave advice on bytecode manipulation, and the attendees of the IBM Research Java Conference gave many useful comments. Thanks also go to Dan Wallach, the anonymous reviewers, and our shepherd Benjamin Zorn.

References

[AFM97] O. Agesin, S. Freund, and J.C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the Symposium on Object Oriented Programming: Systems, Languages, and Applications*, pages 49–65, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, Reading, Massachusetts, 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley Publishing Company, Reading, Massachusetts, 1996.

[KH97] Ralph Keller and Urs Hölzle. Binary Component Adaptation. Technical Report TRCS97-20, Department of Computer Science, University of California at Santa Barbara, December 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley Publishing Company, Reading, Massachusetts, 1997.

[LZ97] Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MSW83] David Moon, Richard Stallman, and Daniel Weinreb. *The Lisp Machine Manual*. AI Lab, MIT, Cambridge, Massachusetts, 1983.

[Obj98] Object Design Inc. ObjectStore PSE Resource Center, 1998. `http://www.odi.com/content/products/PSEHome.html`.

[OHBS94] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-Oriented Programming: Supporting Decentralized Development of Objects. In *The 7th IBM Conference on Object-Oriented Technology*, 1994.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, January 1995.

[Rat98] Rational Software Corporation. Purify, 1998. `http://www.pure.com/products/purify`.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[SG97] Daniel J. Scales and Kourosh Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *The Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *1998 IEEE Symposium on Security and Privacy (to appear)*, May 1998.

Deducing Similarities in Java Sources from Bytecodes

Brenda S. Baker
Bell Laboratories
Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974
bsb@bell-labs.com, http://cm.bell-labs.com/~bsb/

Udi Manber
Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu, http://glimpse.cs.arizona.edu/udi.html

Abstract

Several techniques for detecting similarities of Java programs from bytecode files, without access to the source, are introduced in this paper. These techniques can be used to compare two files, to find similarities among thousands of files, or to compare one new file to an index of many old ones. Experimental results indicate that these techniques can be very effective. Even changes of 30% to the source file will usually result in bytecode that can be associated with the original. Several applications are discussed.

1 Introduction

Java bytecode, particularly in the form of applets, is geared to become the common way to execute programs through the web, and not only by traditional computers. Network computers, even appliances, are expected to be controlled by Java bytecode, which will arrive through the net transparently and for the most part, automatically. There is obviously a great need to be able to control all these programs. There will be problems of security, management of updates, portability, handling preferences, deletions, and so on.

This paper introduces several techniques to help one aspect of these problems. We show that it is possible, given a large set of bytecode files, to identify most of those that originate from similar source code. In addition, an index can be built from any number of bytecode files, such that given a new bytecode one can very quickly find all the old ones that are similar to it. Our tools also can detail in a convenient way just which sections of the files are similar.

Our techniques make it possible to identify bytecode files that came from the same source without knowing the actual source even if significant parts of the source files are different. The files do not even have to be of similar sizes. For example, the original source code files may be different versions of the same program or different programs containing similar sections.

Finding similarities from compiled code is much more difficult than finding similarities in source code or text, because a small change in the source code can produce completely different binaries when they are viewed just as sequences of bytes.

To accomplish these goals, we adapt three tools designed to find similarity in source code and text to work with bytecode files. Siff [24] [1] is designed to analyze large number of text files to find pairs that contain a significant number of common blocks ("fingerprints"), where a block corresponds to a non-trivial segment of code, usually a few lines.

[1] The original name was SIF, but at some unknown point during the development an extra f was added to make it more natural.

Dup [5], searches sets of source files to look for sufficiently long sections that match except for systematic transformations of names, such as variable names, and can deal efficiently with a few million lines of source code. The UNIX utility diff (described in [21]) uses dynamic programming to identify line-by-line changes (insertions or deletions) from one file to another, and is useful for detailed comparison of two files and for automated distribution of patches.

None of the programs mentioned above was designed to deal with binaries. The only work that we know of that may have applied one of these to binaries is .RTPatch [27], a tool for creating patches (differences between files) for updating files. This tool is claimed to work for arbitrary files, even binaries; while no technical description is given of their methods, it seems likely that it applies the diff algorithm to arbitrary bytes rather than to hashes of lines of text. We are not familiar with any work that can find similarities in large number of binaries.

Our adaptations of siff and diff do not work directly on bytecode files, or even on disassembled bytecode files. Bytecode files contain many indices into tables, and values of most indices can be changed by a slight (even one character) change to a Java source file. Therefore, we encode disassembled bytecode files in a "normal form" which takes into account positional values that are less affected by small changes than absolute values of indices. This encoding, based on a technique first used in dup, enables the programs to find the hidden similarity in bytecode files. Since this technique is already incorporated into dup, dup can work on disassembled bytecode files; however, additional simple preprocessing improves its performance.

There are other tools for finding similarities in text or source code, as well. However, tools based on style metrics, such as [7, 19, 25], or data flow graphs [17] would require decompilation of bytecode files in order to be applied. Some other tools based on fingerprints, such as [16, 20, 10], chunks of text [9, 30, 34], or visualization via a graphical user interface [11] may be adaptable to byte code files using the same techniques that we use for siff, dup, and diff.

To search for similar files in a large set of bytecode files, we run siff on the encoded disassembled bytecode files and dup on the preprocessed disassembled bytecode files. We show that combining the output from siff and dup is more effective than either individually at finding similar files while keeping false positives low. We then use dup and diff on the similar pairs to examine the similarities in more detail.

Our programs are fast and can analyze thousands of bytecode files in minutes. A new file can be compared to an index of thousands of existing files in a second or two. The number of false positives is kept to a very small minimum. Our programs are written in C and run under UNIX. Detailed experimental results are given later in the paper.

We foresee several applications for our tool.

program management: When people have numerous Java classes, and get many more on a regular basis, there will be a great need to organize them, often not according to the original "plan." Being able to tell which classes are similar can be very helpful. Sometimes a similarity will identify the source. Knowing that a very similar class is already installed on one's disk may be helpful in deciding what to do with the new class. It can also help with version control, etc. In some cases, especially for programs that perform mostly arithmetic operations, it may be possible to identify programs that implement the same algorithm, even when they are written by independent writers. For example, when we ran experiments on thousands of arbitrary class files, we discovered two MD5 programs with 78% similarity in their bytecode files.

Plagiarism detection: Our approach will identify stolen code if only minor changes are made to it (including any amount of syntactic changes such as changing variable names). However, the advance in sophistication of obfuscators [13, 12], especially for Java, would allow someone to hide any code segment pretty well, and our methods (and very possibly any other method) will not be able to identify it. One may be able to identify that obfuscators were used, on the other hand, which may be good enough (e.g., in a classroom).

Program reuse and reengineering: It will obviously be useful to know which classes are similar when trying to reuse them. Identifying versions is a good example of that.

Uninstallers: If you finally obtain the exact code you wanted, then you probably want to get rid of other code that is very similar.

Security: Detecting that a class is similar to a known

bad class can be very helpful.

Compression: It may be possible to store only differences among classes that are very similar. But first one must detect all such similarities. This could be especially useful for low-storage devices.

Clustering: Small similarities are quite typical among programs written by the same person. Our tools cannot by themselves cluster everything, but they can help a clustering program identify many candidates.

Miscellaneous: There was recently discussion in the press about Sun Microsystems allegedly using a copy of a popular benchmark program directly in its Java compiler. We believe that our program would have discovered that easily and could be used to continuously monitor for cases like that.

The rest of the paper is organized as follows. The next section describes the three similarity tools that we adapt to Java bytecode files. Section 3 describes how we process Java class files to make them suitable for comparisons by dup, siff, and diff. In Section 4 we present experimental results. These include experiments with siff alone, siff and dup together, and diff alone. Section 5 describes related work. We end with conclusions and future research.

2 Tools for finding similarity

This section describes the three tools (diff, siff, and dup) that we adapt to finding similarity in Java bytecode files. All three were designed originally for source or text files.

2.1 Siff

Siff[24] is a program to find similarities in text files. It uses a special kind of random sampling, such that if a sample taken from one file appears in another it is guaranteed to be taken there too. A set of such samples, which are called in siff *approximate fingerprints*, provides a compact representation of a file. With high probability, sets of fingerprints of two similar files will have large intersection, and the fingerprints of two non-similar files will have a very small intersection. Siff works in two different modes: all-against-all and one-against-all. The first mode finds all groups of similar files in a large file system and gives a rough indication of the similarity. The running time is essentially linear in the total size of all files, which makes siff scalable. (A sort, which is not a linear-time routine, is required, but it is performed only on fingerprints, and therefore does not dominate the running time unless the total size of all files is many GBytes.) The second mode compares a given file to a preprocessed *approximate index* of all other files, and determines very quickly all files that are similar to the given file. In both cases, similarity can be detected even if the similar portions constitute as little as 25% of the size of the smaller file (for smaller percentages, the probability of false positives is non trivial, so although siff will work, its output may be too large).

2.2 Dup

Dup [4, 5, 6] looks for similarity of source codes based on finding sufficiently long sections of code that almost match. Dup's notion of almost-matching is the *parameterized match* (*p-match*): two sections of code are a p-match if they are the same textually except possibly for a systematic change of variable names; e.g. if all occurrences of "count" and "fun" in one section are changed to "num" and "foo", respectively, in the other. For a threshold length (in lines) specified by the user, dup reports all longest p-matches over the threshold length within a list of files or between two lists of files. It can compute the percentage of duplication between two files or the percentage of duplication for the cross-product of two lists of files. It can also generate a profile that shows the matches involving each line in the input and a plot showing where matches occur. Dup has been found useful for identifying undesirable duplication within a large software system, looking for plagiarism between systems, and for analyzing the divergence of two systems of common origin.

Using dup, one may choose to base a notion of similarity on the existence of matching sections over a threshold length, on the percentage of common code resulting from these matching sections, or on some combination of the two.

The key idea that enables dup to identify p-matches is to replace tokens such as identifiers by offsets to remove the identity of the identifier while

preserving the distinctness of different identifiers. In particular, the first occurrence of an identifier is replaced by a 0, and each later occurrence of the same identifier is replaced by the number of tokens since the previous one. For example, `u(x,y)=(x>y)?x:y;` and `v(w,z)=(w>z)?w:z;` are both encoded as `0(0,0)=(6>6)?5:5;` because u and v occur in the same positions, as do the pair x and w and the pair y and z; the numbers 6 and 5 represent the number of tokens (single symbols, in this case) between successive occurrences of the same identifier token. More generally, if max, arg1, and arg2 are tokens, the same encoding represents `max(arg1,arg2)=(arg1>arg2)?arg1:arg2;`.

Dup computes longest p-matches via a data structure called a *parameterized suffix tree* (*p-suffix tree*). The p-suffix tree is a compacted trie that represents offset encodings of suffixes of the token string, but only uses linear space because only the offset encoding of the entire input is stored. At each access to an offset, the previous context is used dynamically to determine whether this is the first use of this parameter within the suffix being processed. The algorithms for building a p-suffix tree and searching it for duplication are described in [5, 6].

2.3 Diff

Diff is the oldest tool for finding commonality between files. Given two text files, diff reports a minimal length edit script for the differences between the two files, where edit operations include insertions and deletions. Many algorithms for minimal edit scripts have appeared in the literature. A popular version of diff today is the GNU implementation based on an algorithm of Myers [26]. Because of the quadratic worst-case running time, diff can be slow for comparing large amounts of code with many differences.

Graphical user interfaces such as gdiff (which runs on SGI workstations under UNIX) make it convenient to look at output from diff by aligning identical sections of two files, side by side. Variants of gdiff exist for other platforms; a list is given in [15].

2.4 Combining the tools

The three tools use three very different notions of similarity. In this paper, we show that a combination of the three is more powerful than any of them individually.

For searching large numbers of files, siff and dup can be used together either to broaden the notion of similarity (by taking the union of pairs of files found by siff and dup) or to make it more stringent (by taking the intersection of the pairs reported as similar by dup and siff).

Siff finds similarities that are not found by dup because occasional differences may disrupt the longer matches looked for by dup; on the other hand, dup finds some files that have long matches but aren't reported by siff because the overall percentage of similarity in the files is too low. For example, if a block of code is added to an otherwise unmodified file, the percentage of similarity might fall below the percentage of similarity selected for siff to report, but dup would find that the rest of the file was unchanged.

For more detailed analysis of files found to be similar, dup, diff, and gdiff are useful. Dup provides a list of matching sections of code, a profile showing for each line what other lines match (based on matches over threshold length), and scatter plots for visualization. Diff and gdiff provide an alignment of the two files that enable looking at the differences line-by-line, and are especially effective when the number of differences is very small.

3 Adapting the tools to Java bytecode files

This section describes how we adapt dup, siff, and diff to handle Java class files. A class file, usually obtained by compiling a Java file, is a stream of bytes representing a single class in a form suitable for the Java Virtual Machine [22]. We use the terms "class files" and "bytecode files" interchangeably.

3.1 Information in Java class files

The first items encoded in a class file are a magic number that identifies the file as a class file, the minor version number, and the major version number. A Java Virtual Machine of a particular major and minor version number may run code of the same

major version number and the same or lower minor version number. The version handled currently by our system is major version 45, minor version 3, described in [22].

The remainder of a class file is mainly tables of structures. Of these, the "constant pool" and the method table are important to the design of our tools.

The constant pool contains information about all the constants used in the class, e.g. strings, integers, fields, and classes. This table is very important as many other parts of the class file (including code for methods) contain indices into the constant pool. For example, a reference to a different class includes the index for the string that names that class.

For each method, the method table contains the code and other information such as the number of local variables, exception-handlers, the maximum stack length, indices into the constant pool representing the method name and a string describing the method type, and optional tables aimed at debuggers, such as line number relationships between bytecode and source and information relating local variable names to the code.

3.2 Disassembly of Java class files

The first step in processing a Java class file is to disassemble it. Disassembling a class file requires keeping track at each point of how far the parsing has gotten in the conceptual hierarchy of tables and structures, based on tables in the disassembler derived from [22]. We wrote our own disassembler, although others have been implemented previously; a list appears in [32].

After disassembling the file, we extract the code of each method for further analysis. The disassembled code contains opcodes and arguments for a sequence of assembly-language-level instructions. Figure 1 shows an example of a section from the middle of a disassembled bytecode file, with comments added to identify the numerical opcodes. Since opcodes can have variable numbers of arguments, we put one opcode or argument per line. A character at the start of the line identifies the type of item on the line. Opcodes, indices into the constant pool, indices into the local variable table, signed integers, unsigned integers, and jump offsets are identified by 'o', 'c', 'v', 'i', 'u', and 'j', respectively. Jump offsets are translated from numbers of bytes in the class file to numbers of lines in the disassembled file.

```
o182            #invokevirtual
c106
o153            #ifeq
j+17
o025            #aload
v4
o180            #getfield
c253
o025            #aload
v5
o182            #invokevirtual
c102
o025            #aload
v4
o180            #getfield
c253
o025            #aload
v4
o182            #invokevirtual
c260
o025            #aload
v5
o180            #getfield
c253
```

Figure 1: Disassembled bytecode

Dup can be run on disassembled bytecode if it is provided with an appropriate lexical analyzer, though performance is improved by undoing the jump offsets before running dup. (See the next section.)

Running siff or diff on the disassembled file without further preprocessing does not produce useful information. For example, changing a 4 to a 5 in two places in a 182-line Java file resulted in over 1100 lines of diff output on the disassembled bytecode file, and less than 1% similarity reported by siff.

The reason that siff and diff fail is that indices into the constant table or local variable table may change with slight changes to the Java file, due to additions, deletions, or reorderings of the constant pool and/or local variable table. Such indices typically occur frequently in the code, as in the example of Figure 1. When the files mentioned in the previous paragraph are preprocessed as described shortly, diff reports only changes in two lines (containing opcodes referring to a constant of 5 rather than 4) and siff finds 98% similarity.

3.3 Preprocessing for dup

With an appropriate lexical analyzer, dup can be run on disassembled bytecode files. However, before running dup, it is preferable to undo the jump offsets already present in the disassembled code by changing jumps into a "goto label" form. Dup will compute the offsets itself for the labels. The dynamic way in which dup calculates offsets relative to suffixes means that when two otherwise identical sections of code contain jumps to earlier points and the jumps cross insertions or deletions, these jumps will not cause mismatches, as would happen with a precomputed fixed encoding. Thus, this preprocessing enables dup to find longer p-matches.

Our lexical analyzer for the disassembled bytecode files (without jump offsets) breaks up the input into two classes of tokens: parameter tokens and non-parameter tokens. Offsets are computed for parameter tokens but not for non-parameter tokens. Parameter tokens include indices into the constant pool, indices into the local variable table, labels for jumps, and signed and unsigned integers; the various types of parameter tokens are distinguished so that the offsets are computed separately and tokens of different types will not be matched to each other in parameterized matches. The non-parameter tokens include opcodes.

3.4 Further preprocessing for siff and diff via offsets

Even though absolute values of table indices in bytecode files may change with slight changes to the Java source, there are still hidden similarities in the bytecode files. In particular, the corresponding uses of indices maintain the same *positional* relationship.

Consequently, we use the same offset encoding that is used in dup. In the context of disassembled bytecode, what corresponds to the "identifiers" are the indices into the constant pool or local variable table. (Jumps are already encoded as offsets in the bytecodes.) Siff and diff are then run on the offset-encoded files.

We treat each index into the constant pool or local variable table as a parameter to be replaced by an offset. The first occurrence of each index is encoded as 0, and thereafter each use of an index is encoded as the negative of the number of lines since the previous use (if any) of the same index. Offsets for indices into tables are negative to be consistent with jump offsets, which are negative for a jump to a preceding line and positive for a jump to a later line. The offsets are calculated independently for the constant table and the local variable table. The example of Figure 1 is shown in Figure 2 after calculating offsets for the entire file from which this section was extracted.

```
o182
c-26
o153
j+17
o025
v-10
o180
c-10
o025
v-10
o182
c-26
o025
v-8
o180
c-8
o025
v-4
o182
c-26
o025
v-12
o180
c-8
```

Figure 2: Disassembled bytecode after calculating offsets

This encoding decreases reliance on the absolute value of the indices but preserves the information as to whether indices for different instructions are the same or different. For example, if two files are the same except that indices in one file are always one larger than indices in the other, the encodings of the two files will be identical. The next section describes experiments demonstrating that this encoding enables siff and diff to work effectively.

4 Experiments

4.1 Experiment 1: Random changes

In the first experiment, we took one Java program and made many different random changes to it. These changes included addition of statements (e.g., "newvariable = 43;") in random places, and substitution/deletion of statements (e.g., changing complex conditions in "if" and "while" statements to "i < 1"). We varied the number of changes and the ratio between additions and deletions. We ran these tests on two different Java programs. For each run, we measured the similarities of the source code (using the original siff) and the similarities of the bytecode files. The results, shown in Table 1, consistently show that the bytecode similarities are close to the source code similarities. For each of the two programs we partitioned the tests into three groups according to source similarities: 90-100%, 80-89%, and 70-79%. The results are averages for each group, showing the number of trials, the average similarity for source and bytecode, and the maximal difference between them in all the trials.

Furthermore, we also ran siff on all the variants of the two programs together, and found no false positives.

4.2 Experiment 2: a large set of bytecode files

In the second set of experiments we took 2056 Java bytecode files (from 38 collections of files from many different sources) and ran tests on all of them at the same time (allowing for at least 50% similarity). The goal was to look for similar files from different collections.

Siff reported 634 ordered pairs of files with similarities of at least 50%. We define similarity of two files as the percentage of one file that is contained in the other. As a result, similarity is an asymmetric relation: for example, if a file A is contained in another file B twice its size, then A is 100% similar to B, but B is 50% similar to A. We use ordered pairs here for this reason.

Of the 634 pairs, 591 were between files in the same collection, and 43 were between files in different collections. Next, we use dup to aid in analyzing which of these similar pairs represent interesting relationships.

For the same 2056 Java bytecode files, dup reported 92 ordered pairs of files to have at least one common code section of 200 lines or more, in comparison to the 634 ordered pairs reported by siff. Table 2 shows the breakdown as to how many ordered pairs were reported by siff alone, by both siff and dup, and by dup alone.

The goal of the experiment was to look for similarity in files from different collections (out of the 38 collections we downloaded). The second line of the table gives the breakdown with respect to similarities between files from different collections. The initial analysis was in terms of ordered pairs, since similarities can be asymmetric, but for ease of discussion, the third line shows the corresponding number of unordered pairs.

Of the 9 different-collection (unordered) pairs reported by both siff and dup, we believe that 8 pairs are originally from the same source, based on the pairs of files having the same name. Of the 8 pairs, four pairs are identical files, and four are in the range of 57%-100% similar according to siff and 45% to 97% similar according to dup (based on just the code sections of at least 200 lines that match.)

The remaining different-collection (unordered) pair of files reported by both siff and dup are a pair of programs from different sources (cryptiX, developed by Wolfgang Platzer, and part of JavaFaces, developed by John Thomas) to compute MD5. Siff found them to have 78% similarity (and 86% in the other direction), and dup found them to have a single common code segment of 1336 lines. The common code segment corresponded to 60 identical lines in the Java files. There was no similarity otherwise in the Java files, but siff found additional similarities in the bytecode files. The 60 identical lines are different from the corresponding lines of the MD5 RFC [29], but semantically equivalent. Interestingly, a search of the WWW turned up a third Java program with the same 60 lines. Possibly two of these programs borrowed from the third, or all three copied a description of MD5 other than the RFC.

The different-collection (unordered) pair reported only by dup looks at first glance as if it surely must come from a common source, because dup reported a common code section of 1521 lines, representing 85% of one file and 28% of the other. We didn't have

program	range of source similarity percent	number of trials	average source similarity percent	average bytecode similarity percent	max difference percent
Program 1	90-100	59	93.2	88.7	9
	80-89	76	84.4	78.7	9
	70-79	35	75.9	71.9	7
Program 2	90-100	17	92.9	91	9
	80-89	25	83.4	80.2	8
	70-79	6	76.2	74.2	5

Table 1: Similarity found by siff for corresponding Java files and bytecode files

	siff only	both	dup only
ordered pairs	552	82	10
ordered pairs, different collections	25	18	2
unordered pairs, different collections	23	9	1

Table 2: Similarities reported by siff and dup for 2056 bytecode files

the java source to compare. Upon inspection of the bytecode files, however, the common code turned out to be the initialization of an array of size 256. Since the stored values (retrievable from the constant pool of the bytecode files) appeared unrelated, the similarity is probably coincidental, merely an artifact of how the compiler generates code for a series of 256 array assignments.

For the pairs reported only by siff, the Java source was not available. However, almost all result from comparing a very small file (500 bytes or less in most cases) to a large file; the small file was found to be similar to the large one, but not the other way around. In addition, the names of these pairs indicate that the purposes of the files are unrelated. We conclude that these are false positives. The exceptions were matches whose names included H or V (apparently for horizontal and vertical); the H/H and V/V pairs were reported by both siff and dup, but the H/V and V/H pairs only by siff. The names of another two pairs of matching files related to buttons and checkboxes, but the remaining pairs of files appear to be totally unrelated based on the subject matter indicated by the names.

To summarize, we found many similarities and very few false positives by combining the information from dup and siff. The different-collection pairs reported by both dup and siff all appear to be valid instances of similarity. The one reported only by dup was not; some of the ones reported only by siff were not, especially for small files. Overall, the number of false positives was very small: at most 18 out of more than 2 million pairs.

4.3 Experiment 3: False negatives

The first two experiments indicate that our tools can effectively discover similarities while minimizing false positives. But the question of false negatives – that is, how many similar pairs we missed – remains unresolved. There were none in the first experiment, but it was too limited.

Since there are no other tools to compare to, there is no guaranteed way for us to measure false negatives. Nevertheless, we believe that the following "blind" experiment gives a reasonable indication.

We asked a friend who was familiar with the goals of our work, but not with the techniques we use, to randomly pick 10 programs from a set of 765 java programs (a subset of the 2056 programs for which we had the source), make random changes to them, compile them (possibly under a different version of the compiler) and give the set of bytecode files in a random order. We then ran siff and dup to see how many of his changes we can detect.

To make the test even more blind, our tester actually made changes to 12 programs, and added to our original test some additional programs (as well as removed some programs). siff running with a thresh-

old of 65% similarity discovered 9 of the 12. Dup, with a threshold of 100 common lines, discovered 8. Together, they discovered 10. All 12 were found by siff when run with a 25% similarity threshold and by dup when run with a 50-line threshold. (On the other hand, decreasing the threshold to 25% increases the number of ordered pairs siff finds among the 2056 original files from 634 to 1430. Still reasonably small compared to the total of over 4 million possible pairs, but clearly the number of false positives grows when the threshold is decreased.)

Out of the two programs that were missed by both siff and dup, one is particularly interesting. It involved relatively few changes to the source, but they made the bytecode file very different. On close inspection, we found that the main culprit was a move of a segment of code, which resulted in a bytecode file with many jumps around the relocated code. For siff, the offsets of all these jumps were affected by the relocation, resulting in different fingerprints. For dup, the relocated code broke up long matches in both places, which mattered since the bytecode files were small - only four times the threshold length.

To validate the usefulness of the offset encoding for diff, we ran diff on the disassembled code with and without the offset encoding. The results are shown in Table 3. The pairs varied in file size and how many changes were made. To get a relatively size-invariant measure, we use the length of the diff output (in lines) divided by the sum of the file sizes, for each type of file. (Pair 6 is special: different versions of the JDK compiler were used to generate two class files from the same java file. Consequently, no value is given for this measure for the java file.) Our second measure is the average length of blocks of identical lines reported by diff. The last column contains the sum of the sizes of the offset-encoded disassembled files, which is the same as the sum of the file sizes without the offset-encoding.

For most of the pairs, the data in Table 3 show a significant improvement from using offset encodings for diff. In a few instances, the values are about the same with and without offsets. Pair 11 is the only instance where diff found significantly less similarity with offset encodings, but just for the second measure. Note that pairs 3, 10, and 12 were the ones not discovered by siff with a 50% threshold, as discussed above, and pairs 9-12 were the ones not discovered by dup with a 100-line threshold.

To validate our approach of using offset encodings for siff, we also ran siff on the disassembled code without applying the offset encoding. With a 25% threshold of similarity, siff discovered only three of the 12 pairs (4, 5, and 10), a much worse performance than that described above for the offset encoding.

4.4 Running times and scalability

For the 2056 files, siff used 41 seconds of elapsed time and 4 seconds of user time while dup used 1 minute 55 seconds of elapsed time and 1 minute 7 seconds of user time (running under IRIX 5.3 and using one of 12 150 MHZ IP19 Processors, with data cache size 16 Kbytes, instruction cache size 16 Kbytes, secondary unified instruction/data cache size 1 Mbyte, and main memory size 1280 Mbytes). Encoding the 2056 bytecode files took 7 minutes of elapsed time for siff and 8 minutes (independently) for dup. (Most of the preprocessing could be shared.) Siff also enables the creation of an index so that new files can be compared with an index of files processed earlier. Comparing one 50K bytecode to all the 2056 files in the index (within 50% similarity) takes a couple of seconds for encoding the new file and thereafter the processing by siff is essentially instantaneous (0.2 user time and between 0:00 and 0:01 elapsed time). The size of the index depends on the amount of sampling done and the precision of results; for the experiments used here the index occupied about 5% of the total size of all files.

Dup is much greedier in space than siff, and consequently will not scale to processing huge numbers of bytecode files at one time. If the goal were to process an enormous number of bytecode files, say to provide a registry service for the World Wide Web as has been proposed for text files [9], then siff should be used to screen for similarities that should be checked subsequently by dup.

5 Related Work

Java bytecode files are relatively easy to decompile into Java. Programs can be rewritten in structured form by analyzing the flow of control, and the names of classes, fields, and methods are available from the class file. In fact, at least four bytecode decompilers have been implemented: Mocha (by Hanpeter Van

pair	diff-size/total-lines as %			ave. ident. block size in lines		sum of file sizes in lines
	java	offset	no-offset	offset	no-offset	offset = no-offset
1	13	6	57	185	3	2740
2	14	17	68	18	2	1040
3	30	24	61	8	2	2098
4	14	14	21	173	20	1603
5	59	40	42	10	10	988
6	-	2	34	198	5	6041
7	19	13	32	22	6	8789
8	23	24	59	36	12	941
9	14	8	45	61	4	265
10	24	32	31	7	8	662
11	50	55	56	7	24	428
12	18	41	67	6	2	1037

Table 3: Results of running diff with and without the offset encoding

Vliet, now deceased, and according to [31], taken over as part of Borland's JBuilder [8]), WingDis [33], DejaVu [18], and Krakatoa [28]. A discussion of these appears in [14]. These produce quite readable code except for variable names, and even those may be available (perhaps inadvertently) from the class file if the compiler generated optional information aimed at debuggers.

Because class files are relatively easy to decompile readably, Java applications are potentially vulnerable to theft, even if they are distributed only as bytecode files. An unscrupulous person could use one of the decompilers mentioned above to decompile the bytecode files of an application, modify the resulting source, recompile into bytecode files, and distribute the bytecode files.

In fact, a number of "obfuscators" have been written to make decompilation less successful. These include Crema (by Van Vliet, and like Mocha, also reportedly taken over by Borland's JBuilder[8]), HoseMocha, by Mark LaDue, HashJava (now renamed SourceGuard [1]), by K.B. Sriram, and Jobe, by Eron Jokipii; for discussion of these, see [23, 32]. Early obfuscators either changed symbol names or added no-ops to bytecode files in such a way that particular decompilers crashed. New obfuscators (e.g., [13, 12]) employ much more sophisticated techniques, some of cryptographic strength, to change the appearance of code. These techniques not only make it harder to decompile, but they also make it possible to change the appearance of bytecode files that come from the same source. Our methods will not be able to defeat such techniques, although it may be possible to detect that they are used.

Another technique for enabling detection of stolen code is steganography, the art of hiding information such that the information can later be detected [2]. For object code, insertions of extra no-op sequences of instructions would slow it down, but Intel reports that it has successfully replaced code fragments by equivalent ones to customize security code [3]. However, this is not common practice.

For Java bytecode files, the single Java Virtual Machine means that the variety of platforms is not an issue. Multiplicity of compilers can be dealt with by having the owner of the code compile the source files with each of the available Java compilers and compare the resulting bytecode files with the bytecode files that were suspected to be stolen. Currently, the number of different compilers is not large, so this is manageable. Code compiled by different releases of the same compiler will probably still be very similar, although not necessarily identical. In our experiments we found that, given the same source files, jdk 1.0.2 and jdk 1.1.x generated class files that rarely differed in any significant way.

6 Conclusions and Future Work

Our experiments have validated our approach by showing that our tools are able to deduce similarity in Java source from similarity in the bytecode files. We can make the rate of false positives very

low while keeping false negatives reasonably minimal. Our positional encoding proves to be a very powerful technique. Since the three tools siff, dup, and diff are based on different techniques, one may detect similarity when another one misses it. Local modifications within sections of code will reduce the percentage of similarity found by siff, but dup will still find long matches in the unchanged sections. Name changes of variables would not be a problem since the compiler turns variable references into table indices, which we have shown our methods to handle despite change in absolute values. Reordering major sections of code, e.g. by changing the order of methods, will have little effect on siff and dup, though it will greatly affect diff.

A major advantage of looking at Java class files rather than at binaries for other programming languages is that there is just one version of the Java Virtual Machine for all platforms (or at least that is Sun's intention), while binaries of other languages are different for different platforms. Extending our techniques to binaries is a natural step to take, although binaries present several additional problems: They are strongly tied to the architecture, there are many compiler and even more optimization and code restructuring programs, and the information in binaries as not as well organized as in bytecode files. Nevertheless, we believe that it will be possible, at least in some degree, to identify similar binaries.

It would be helpful to have a graphical user interface that links the output of siff and dup to a decompiler, allowing a developer to see the evolution of two similar programs clearly.

7 Acknowledgements

Peter Bigot was our blind tester, and we thank him for a thankless job well done.

8 Availability

Contact Brenda Baker, bsb@bell-labs.com, for information about licensing the software from Lucent Technologies.

References

[1] 4thpass Software Corporation. Protect your Java investment. `http://www.4thpass.com/`, Apr. 7, 1998.

[2] Ross Anderson and Fabien Petitcolas. On the limits of steganography. In *IEEE J-SAC*, to appear.

[3] D. Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.

[4] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[5] Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, Feb. 1996.

[6] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Computing*, 26(5):1343–1362, Oct. 1997.

[7] H.L. Berghel and D.L. Sallach. Measurements of program similarity in identical task environments. *SIGPLAN Notices*, 9(8):65–76, August 1984.

[8] Borland. Jbuilder. `http://www.borland.com/jbuilder/`, Oct. 23, 1997.

[9] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 1995.

[10] Andrei Broder, Steve Glassman, Mark Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, April 1997.

[11] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages 1998*, Chicago, IL, May 1998.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, Jan. 1998.

[14] Dave Dyer. Java decompilers compared. *JavaWorld*, July 16, 1997. `http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html`.

[15] Luis Fernandes. Are there any apps that can display files in parallel, highlighting (in color) the differences between them? `http://www.ee.ryerson.ca:8080/~elf/xapps/Q-XI.html`, April 3, 1997.

[16] Nevin Heintze. Scalable document fingerprinting. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Nov. 18-21, 1996.

[17] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, June 1990.

[18] Innovative Software. OEW for Java. `http://www.isg.de/OEW/Java/`, Oct. 2, 1997.

[19] H.T. Jankowitz. Detecting plagiarism in student PASCAL programs. *Computer Journal*, 31(1):1–8, 1988.

[20] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proc. International Conf. on Software Maintenance*, pages 120–126, 1994.

[21] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachusetts, 1997.

[23] Qusay H. Mahmoud. Java tip 22: Protect your bytecodes from reverse engineering/decompilation. *JavaWorld*, Jan. 2, 1997. `http://www.javaworld.com/javatips/jw-javatip22i.html`.

[24] Udi Manber. Finding similar files in a large file system. In *Proc. 1994 Winter Usenix Technical Conference*, pages 1–10, Jan 1994.

[25] T.J. McCabe. Reverse engineering, reusability, redundancy: the connection. *American Programmer*, 3(10):8–13, Oct. 1990.

[26] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[27] PocketSoft. .RTPatch Professional, Feb. 23, 1998. `http://www.pocketsoft.com/products.html`.

[28] Todd Proebsting and Scott A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *USENIX Conference on Object-oriented Technologies and Systems*, June 1997.

[29] Ronald Rivest. The MD5 message digest algorithm. RFC 1321, Apr. 1992. `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1321.txt`.

[30] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM International Conference on Digital Libraries (DL'96)*, March 1996.

[31] Eric Smith. Mocha, the Java decompiler. `http://www.brouhaha.com/~eric/computers/mocha.html`, Dec. 28, 1997.

[32] Thomas Traber. Tools for working with bytecode: Bytecode assemblers, disassemblers and dumps. `http://www.oasis.leo.org/java/development/bytecode/00-index.html`. Part of Java Oasis, `http://www.oasis.leo.org/java/00-oasis.html`.

[33] Wingsoft. Wingsoft Products Information. `http://www.wingsoft.com/products.shtml`.

[34] Tak Yan and Hector Garcia-Molina. Duplicate removal in information dissemination. Technical report, Stanford Computer Science Dept., 1995. submitted for publication.

Transformer Tunnels: A Framework for Providing Route-Specific Adaptations

Pradeep Sudame B. R. Badrinath

Department of Computer Science
Rutgers University
Piscataway, NJ 08855
{sudame,badri}@cs.rutgers.edu

Abstract

In a network using links with diverse properties, a packet flow that is fine tuned for some links (by selecting a proper packet size, transmission rate, encryption method, etc.) may be inappropriate for other links. Ability to change the flow properties over segments of the network allows flows with different characteristics to coexist; making it possible to adapt to diverse link properties. Application-specific adaptation mechanisms (such as proxies) do not force adaptations on every packet flowing over the link and are therefore insufficient for this purpose. We propose the concept of *transformer tunnels* that force adaptations on all the packets flowing through them. Transformer tunnels can coexist with proxies because the adaptations provided by both are independent of each other. Transformer tunnels provide adaptations by means of *transformation functions*. By attaching various transformation functions to such tunnels, we can efficiently fine tune the flow properties. We also provide an API for developing transformation functions. We have implemented transformer tunnels and have used them in our wireless network. In this paper, we present the effects on mobile hosts that use this mechanism to transform flows over the last-hop link for reducing losses during handoffs, and for improving the link utilization.

1 Introduction

The availability of various networking technologies leads to a flow of packets over links with diverse properties. Mobile hosts increase this diversity by using wireless technologies such as WaveLAN [35], CDPD [2], Metricom Ricochet [25], Satellites, cellular modems, and so on. Sometimes such wireless networks are used as the backbone technology as well [17]. In such networks, some links are cheap, fast, reliable, and secure; whereas some links are expensive, slow, lossy, and insecure. Thus, packet flow that is fine tuned for some links (by selecting a proper packet size, transmission rate, encryption method, etc.) may be inappropriate for other links. For example, sending large packets over reliable links improves throughput, whereas sending large packets over lossy links increases the probability of retransmissions. Further, a user can have access to multiple devices such as PDAs, laptops, cellular phones and can connect using different networking technologies at different times and different places (as in overlay networks [7]). Such dynamic changes makes the fine tuning of the flow difficult.

A mobile user invariably encounters a heterogeneous network consisting of segments with diverse properties (bandwidth, asymmetry, reliability, etc.). We need the ability to modify the packet flow over various segments of a route for adapting to such properties. For such adaptations, we propose the concept of *transformer tunnels*. Transformer tunnels transform the packet flow to provide application-transparent, route-specific adaptations. Route-specific adaptations, unlike adaptations that are forced on all hosts using a link, allow different hosts using the link to simultaneously request different adaptations. The adaptations depend on the transformation functions attached to the tunnel. We have implemented the transformer tunnels and have used them over our wireless network to provide adaptations for wireless links. In this paper, we show how mobile hosts can use transformer tunnels to change packet flow over the last-hop link. We also provide an API for adding new transformation functions to the system. Using this API, we have implemented some transformation functions such as encrypting data to provide security, sending packets in bursts to allow energy efficient operations, combining small packets and compressing data over slow links to improve link throughput, and so on.

The paper is organized as follows. Section 2 explains the concept of transformer tunnels, and how mobile hosts use such tunnels to achieve adaptation over the last-hop link. Section 3 details the tunneling mechanism. Section 4 describes the transformation functions that we built and tested. It also describes some other functions that can be built within the same framework. Section 5 describes the

[1]This research work was supported in part by DARPA under contract numbers DAAH04-95-1-0596 and DAAG55-97-1-0322, NSF grant numbers CCR 95-09620, IRIS 95-09816 and Sponsors of WINLAB.

experiments we performed to evaluate the transformer tunnels. Section 6 describes the related work. The paper concludes with our plans to extend this work on transformer tunnels.

2 Transformer Tunnels

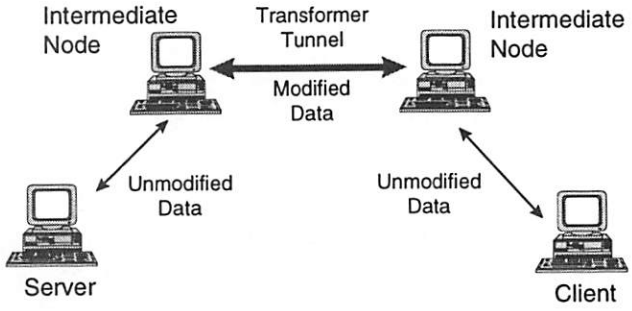


Figure 1: Transformer tunnel

Packet flow over a network segment is modified by placing a transformer tunnel between two end nodes of the segment. Figure 1 shows how packets undergo transformations at the end points of the tunnel. Packets entering the tunnel are modified — either by changing the content, or by changing the way they are transmitted — to fine tune the flow according to the segment properties. Original packets are restored at the other end of the tunnel. Adaptation is thus achieved without affecting rest of the network.

Adaptation over the last-hop link for mobile hosts is achieved by placing similar tunnels over the last-hop link. Even on the same last-hop link, different hosts can request different adaptations. As an example, Figure 2 shows three mobile clients, all requiring different adaptations. Client 1 requires data encryption for its sensitive data because wireless links can be easily snooped upon. Client 2 needs bursty transmission so that it can conserve energy by putting its network interface in sleep mode. At the same time, client 3 requires both the adaptations. This is achieved by establishing transformer tunnels between the base station and each of the clients. The base station then acts as a *transformation agent* for the clients. The transformation agent can be placed anywhere on the network as long as it can intercept the client's packets. In our network, the base station is the logical choice for transformation agents.

Use of proxies has been suggested in the literature for similar adaptations [15, 26, 37]. Transformer tunnels are not a replacement for proxies; rather they supplement proxies. Transformer tunnels differ from proxies in three ways.
Layer of operation: Proxies usually operate at the application layer. Transformer tunnels, on the other hand, provide a low-level application-transparent adaptation. Thus proxies and transformer tunnels can coexist. Proxies can perform application-specific filtering of data (for example, dropping frames in a video stream), whereas transformer tunnels can provide link-specific optimization of the flow (for example, deferring packets for a mobile host till it is in range of an infostation [16]).
Mandatory adaptation: High-level proxies operate on streams associated with a particular application or an application-layer protocol like HTTP; whereas transformer tunnels force adaptations on all the packets passing through them. This feature is required for link dependent adaptations that need to control the way packets are transmitted (for example, making a flow bursty).
End-to-end semantics: Transformations performed by transformer tunnels are hidden from higher layers of the protocol stack. So, if the protocols being used have a notion of a connection (TCP, for example), then unlike proxies, transformer tunnels do not affect the end-to-end semantics of such connections.

2.1 Tunneling Mechanism

Applications configure transformer tunnels by attaching various transformation functions to them. The tunnels apply these functions to all the packets passing through them and send the modified packets over the network. The tunnels support composition of transformation functions. For example, small compressed packets can be combined in larger packets by another transformation function to improve the link utilization. Depending on the link conditions, a mobile host can decide what features are required and can ask the base station to provide an appropriate transformer tunnel. Thus, whenever the link conditions change, the tunnel can be reconfigured. This can be achieved by combining the transformer tunnels with our work on exposing link conditions to higher layers of the protocol stack [32].

Depending on the transformation functions attached, a transformer tunnel modifies incoming packets. The modified packets contain sufficient information so that they can be restored at the other end of the tunnel. As with any other tunnel, a transformer tunnel uses the point-to-point address of the tunnel as the packet's destination address. The origi-

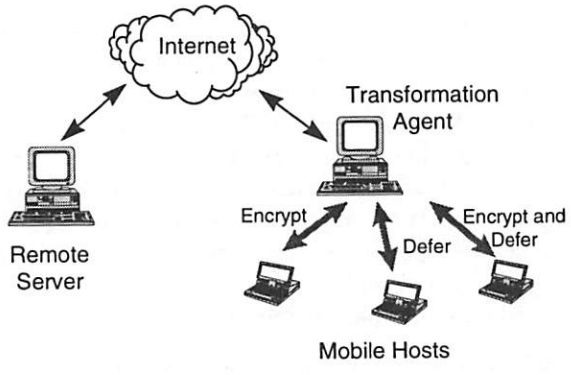


Figure 2: Transformer tunnels on last-hop links

nal destination address, if different, is stored in the data portion of the packet. Some metadata (list of functions to apply to restore the original packet) is also added to the packet.

Transformer tunnels provide a way to build a restricted form of active networks [34]. In case of transformer tunnels, the packets cannot be transformed at intermediate nodes. All the transformations are performed at the end points of the tunnels. This restriction allows an efficient implementation and is still powerful enough to deal with the problems introduced by mobility.

Transformations performed by the transformer tunnels have to be undone at the other end. For this purpose, once the transformations are performed, the transformer tunnel changes the protocol field in the IP header to IP_XFORM. At the other end of the tunnel, the protocol handler for IP_XFORM removes all the metadata from the packet, performs the inverse transformations, restores the original protocol, and puts the packet back in the IP input queue. Some transformations do not change the packet contents (incompressible packets, delayed packets, etc.) and hence do not add any metadata to the IP header. If none of the transformations add any metadata, the packet is delivered with the original protocol, thereby avoiding any overheads at the other end of the tunnel.

A simpler approach would have been to use IP encapsulation. The source address in the new IP header (the outer header used for encapsulation) is not used by the receiver to decapsulate the packet. Moreover, if the the point-to-point address of the tunnel is same as the destination address, the new IP header will have the same destination address. Therefore, to reduce the number of bits used, we modify the original IP header. We add the original protocol number (2 bytes), a flag indicating whether the original destination address is same as the point-to-point address of the tunnel (1 byte), and if required, the original destination address (4 bytes) to the metadata. IP encapsulation would require 20 bytes instead of 7 bytes used by our scheme. (We need just 3 bytes if the destination address is same as the point-to-point address of the tunnel.) Therefore, we save at least 49 μsec on a 2-Mbps WaveLAN.

2.1.1 Adding New Transformation Functions to the System

Transformer tunnels do not assume a fixed set of transformation functions. The transformation functions can be arbitrary as long as the mobile host and the base station agree on them. We provide an API to add new functionalities (using the support for loadable modules provided by Linux) to the system. Any module defining a new functionality has to register the transformation function, and an optional private *ioctl*, with the system by calling the following function.

```
int add_tunnel_func (unsigned char id,
    int (*func) (struct device *dev,
        struct sk_buff **skb,
        struct stunnel_info *info),
    int (*ioctl) (struct device *dev,
        int cmd, void *data, int len));
```

id: unique id for the functionality
func: the transformation function
ioctl: private ioctl for the functionality

The transformation function *(func)* is called with three parameters. The first parameter *(dev)* is the tunnel device. The second parameter *(skb)* is the entire packet with all the headers. (Linux uses *sk_buff* structure to pass packets around.) The third parameter *(info)* must be filled in by *func* with the id of the inverse transformation that should be performed at the other end to restore the packet.

The transformation function has to return either of the two values: PACKET_SENT or PACKET_NOT_SENT. In general, the function will just change the packet contents and let the tunnel handle the actual transmission by returning PACKET_NOT_SENT. Functions that need to control the way packets are transmitted should return PACKET_SENT.

Some modules may want to provide additional *ioctls*. For example, a module that provides buffering requires an *ioctl* to flush all the buffered packets. Other modules may not need any *ioctls*. The corresponding parameter can be *null* in such cases.

While unloading the modules, the functionality has to be removed from the system by invoking the following function.

```
int remove_tunnel_func (unsigned char id);
```

id: id specified when loading the module

2.1.2 Attaching Transformation Functions to Tunnels

Applications attach transformation functions to a transformer tunnel using the following *ioctl* call for the tunnel device.

```
sd = socket (AF_INET, SOCK_DGRAM, 0);
struct ifreq rq;
rq.ifr_name = <tunnel name>;
rq.ifr_data = <id for the function
        to be added, priv>
ioctl (sd, SIOCATTACHFUNC, &rq);
```

The parameter *priv* is used for passing information to the transformation function. For example, an encryption function may use this as the encryption key. A new SIOCDETACHFUNC *ioctl* is used to detach transformation functions from the tunnel.

3 Establishing and Configuring Transformer Tunnels

A transformer tunnel is established like any other tunnel: by specifying the local host's IP address as one end of the tunnel and the remote host's IP address (point-to-point address) as the other end. The tunnel transforms all the packets that are given to the tunnel device, stores the original destination address and the original protocol in the body of the packet, and replaces the destination address with the address of the other end of the tunnel. It then performs a routing-table lookup and prepares the packet for delivery. Normal IP routing then reroutes the packet to the new destination address. In rest of the paper, discussion about transformer tunnels is restricted to tunnels over the last-hop link for mobile hosts. Nevertheless, the discussion is valid for any other transformer tunnel.

A transformer tunnel for a mobile host is established between the mobile host and its base station. All packets destined for the mobile host are forced to go over the tunnel by adding an entry to the routing table at the base station. Once established, the tunnel has to be configured (using the API described in section 2.1.2) to specify the required transformations. The transformation functions are applied in the order in which they are attached to the tunnel.

To allow the mobile host to control the tunnel configuration, we have provided a *tunnel manager* at the base station. Whenever the mobile host requests some adaptation on the last-hop link, the tunnel manager establishes and configures a transformer tunnel accordingly. Moreover, whenever the mobile host moves to a new location, it informs the tunnel manager at the previous location. The tunnel manager then changes the end point of the corresponding tunnel to the new location, forwards all the pending packets, and closes the tunnel. This is a client-server architecture where the client (the mobile host) can remotely configure tunnels at the server (the base station). Thus, the mobile host can add or delete transformation functions at any time. Typically, such a decision will be taken by the mobile host when conditions change. For example, when operating on battery power, it may request that the packets be delivered in bursts so that the network interface can be put in sleep mode in between the bursts. It may then request removal of this feature when operating off an automobile battery. For certain adaptations, the mobile host may have to perform additional functions. For example, if the mobile host requests bursty flow, then it has to monitor the flow, and if the inactivity exceeds a certain threshold, it has to put its network interface in sleep mode.

The tunnel manager has the following features.

Reliability of the adaptation requests: Whenever the tunnel manager receives a request, it sends back a reply indicating whether the action was successful. It is the mobile host's responsibility to ensure that the request is reliably sent to the base station. If it fails to receive the reply, it should retransmit the request. If the reply from the tunnel manager is lost, then retransmitting the request may reconfigure the tunnel with the same parameters. However, reconfiguration leaves the behavior of the tunnel unchanged and hence is harmless.

```
Initialization code
    delayed_packet = null;

reassemble (p) {
    if (p is large) {
        send delayed_packet; send p;
        delayed_packet = null;
        return PACKET_SENT;
    }
    if (delayed_packet exists) {
        if (p and delayed_packet can be combined) {
            combine and send;
            delayed_packet = null;
            return PACKET_SENT;
        } else {
            send delayed_packet;
            delayed_packet = p;
            set timer; return PACKET_SENT;
        }
    } else {
        delayed_packet = p;
    }
}

Timer expires:
    send delayed_packet; delayed_packet = null;
```

Figure 3: Reassembly algorithm

Ability to bypass the transformations: Transformer tunnels do not transform the reply packets sent by the tunnel manager. (Otherwise the reply may be delayed if the tunnel transformation function requires so.) This is achieved by specifying that the packets associated with the socket that the tunnel manager uses should not be modified by the tunnel. Other applications that need to bypass some of the transformations can make similar requests. Such requests are made using an *ioctl* call for the tunnel device.

Soft state: Mobile hosts can move away without proper deregistration. The base station cannot keep the tunnel open indefinitely because it has finite resources. The mobile host thus has to renew the request periodically. If the tunnel is not renewed periodically, the base station removes the tunnel. In other words, the base station maintains soft-state information about the transformation functions required by the mobile host. The state is regenerated by periodic renewals.

4 Transformation Functions

The API provided for developing transformation functions takes away developer's responsibility of dealing with devices. New transformation functions can therefore be developed easily. Using this API, we have implemented five transformation functions.

Reassembly: A packet flowing from a fixed host to a mobile host typically traverses links with different MTUs (Maximum Transfer Unit). To eliminate fragmentation, TCP sometimes uses path-MTU–discovery mechanism and selects the smallest MTU along the path as the TCP segment size. (In practice, many TCP implementations select a pessimistic segment size of 536 bytes to avoid the overheads of path-MTU discovery.) Some wireless devices, such as WaveLAN, have a large MTU (1500 bytes). A narrow link along the path means that packets much smaller than the link MTU will be sent over the wireless link. The reassembly transformation function combines such packets to improve the link utilization and reduces cost if the users are charged on a per-packet basis (as in CDPD networks) for their network usage. The combined packet requires just one link-layer header, thereby offsetting the extra bits sent as metadata. Moreover, reducing the number of packets results in lowered contention over broadcast links.

The reassembly function uses a mechanism similar to TCP delayed ACKs. Every small packet is delayed by a small amount of time. If another small packet arrives in this interval, both the packets are combined. Otherwise, the packet is sent as is. This reassembly mechanism is different from IP reassembly (performed after IP fragmentation). A reassembly function combines small packets, along with their IP headers, to get a larger packet. The mobile host regenerates all the original packets when it receives such a reassembled packet. Thus, this mechanism can be used even for protocols that honor message boundaries (for example, UDP).

We use the reassembly algorithm shown in Figure 3. A reassembly tunnel maintains a single packet buffer for every mobile host that requests this adaptation. The maximum time for which packets are delayed is a parameter specified by the mobile host. The mobile host has to decide the maximum delay it can tolerate. A larger delay value increases the probability of two small packets being combined to create a larger packet. It is possible to extend the strategy to combine more than two packets if more delay is tolerable. To simplify the implementation we combine at most two packets.

Energy savings: Wireless devices usually have power-saving features that are built in the hardware. Software strategies to take advantage of these features are gaining importance [24]. We demonstrate a simple energy-saving strategy by using the transformer tunnels.

For devices like WaveLAN, energy consumed while waiting for a packet is about seven to eight times higher than the energy consumed when the card is in sleep mode. Thus the amount of time the card can be put in sleep mode determines how much energy can be saved [31]. The mobile host may put its WaveLAN card in sleep mode only if it knows when *not* to expect packets. This knowledge can be provided by making the traffic over the wireless link bursty and by informing the mobile host about the time slots when the bursts are sent.

In our implementation, the base station buffers all packets for the mobile host whenever the energy-savings adaptation is requested. The base station also sends out periodic beacons that contain a list of mobile hosts who have a packet pending. The mobile host periodically wakes up to listen to these beacons. If it has a pending packet, it requests that the packet be sent ("card spin-up"). In response, the base station sends all the buffered packets for this host and removes the energy-savings transformation function from the tunnel. The mobile host then stays awake. Based on some inactivity threshold, it decides to enter the energy-savings mode again ("card spin-down"). This "spin-up–spin-down" mechanism has already been used for CDPD links [23]. The energy-savings transformation function allows any other device to use the same mechanism without modifying the corresponding device driver, but is useful only when energy required to power up and power down the device is low (for example, WaveLAN [31].)

The mobile host can sleep for as long as it wants if the base station can buffer any number of packets. However, the buffer space at the base station is limited. Therefore, the mobile host has to wake up periodically. (Mobile-IP implementation [11] requires that the mobile hosts renew their registration every 5 seconds. So in our implementation, a mobile host cannot sleep for more than 5 seconds.) Moreover, the beaconing interval at the base station should be small enough to avoid buffer overflow during the interval.

Buffering: Whenever a mobile host performs a handoff, some packets are lost. These losses can be reduced by buffering the last few packets and forwarding them to the mobile host's new location [9].

A buffering tunnel acts like a write-through buffer that buffers the last few packets sent to the mobile host. Whenever the mobile host experiences a burst loss, which may be due to a handoff, it asks the tunnel manager to flush the buffer (retransmit the last few packets). We have modified the mobile-IP code at the mobile host so that it informs the tunnel manager at the previous base station whenever a handoff occurs. In response, the tunnel manager changes the end point of the tunnel to point to the new base station and forwards the buffered packets.

Encryption: Wireless links are more susceptible to snooping. Some applications may want to encrypt their data over

Transformation Function	When to use	Side effects
Reassembly	large MTU, cost per packet	increases latency for isolated packets, large packets more likely to be lost on noisy links
Encryption	insecure links	processing overheads
Compression	slow links	overheads for incompressible data
Buffer	mobile clients	requires buffers at the base station
Energy Savings	power constraints	increases latency for first few packets in idle mode

Table 1: Currently implemented transformation functions

such links. A transformer tunnel with a simple encryption function may be used here.

Depending on the security requirements, various encryption methods can be used. To demonstrate the feasibility of such a transformation function, we have used a simple XOR function to encrypt data [33]. A more secure method can be used where the mobile host and the base station share a secret key. This secret key can be established (by using public key encryption) when the mobile host requests encryption over the link. If required, a sophisticated IP-encryption mechanism [1] may be used. We do not deal with the issue of key management as it is orthogonal to the mechanism of transformer tunnels. Once the key is obtained, it can be passed as an argument to the encryption function (by using the API described in section 2.1.2).

Compression: On slow links, the time for transmitting packets is large and compressing packets before transmission improves performance. Header compression techniques [10, 21] have been proposed to improve performance for such links. On very slow links, compressing the data portion leads to even more gains [28, 29]. Compressing data is not useful on fast links, because the compression and decompression overheads offset the savings obtained by sending fewer bits. On slow links, however, a fast compression function (where the time per byte for compression and decompression multiplied by the bandwidth is less than the fraction of bytes saved) leads to improved performance.

We have used a simple compression function provided by the minilzo library [27]. If the packet is incompressible, we send the original packet without any modifications. The LZO compression method is fast enough to be useful even on WaveLAN. For slower devices like CDPD, it will lead to more gains. To increase the gains further, a more expensive compression function can be used on slow devices.

The conditions under which these transformations are useful are enumerated in Table 1. Many other transformation functions can be built in the same framework. We describe some of them here. These functions have not been implemented yet in the transformer tunnels framework.

Snoop TCP: Losses on wireless links are usually due to noise on the link. TCP interprets all losses as a symptom of congestion and slows down the transmission. Snoop TCP [6] deals with this problem by having faster retransmissions of lost packets on the wireless link. Transformation tunnels provide a simple framework for implementing Snoop TCP.

Dealing with asymmetry: Asymmetric links (where the uplink bandwidth is small as compared to the downlink bandwidth) result in poor TCP performance [12, 5]. A slow uplink results in underutilization of the fast downlink because TCP senders decide the transmission rate based on the frequency of incoming ACKs.

A transformation function that suppresses a few TCP ACKs reduces the load on the uplink [5]. Dropped ACKs do not affect the reliability of the protocol because TCP ACKs are cumulative. However, ACK filtering leads to bursty transmission from the sender [5]. To alleviate this problem, the base station can regenerate filtered ACKs and can send them to the sender at a steady rate [4].

Support for proxies: Transformation functions at the link layer are unaware of the semantics of the data. Proxies can do better filtering based on the data-type–specific operations. A module can be written that provides support for developing proxies. If required, this module can propagate packets to some filter application. Other packets can be sent over the link without changes. Application filters can take appropriate actions and send the filtered packets back onto the network. Such a module would be similar to the "low-level proxies (LLP)" [37] for application-independent adaptation.

5 Evaluation

This section describes the experiments we performed to evaluate the transformer tunnels. For all our experiments, we used a wireless LAN configuration. The wireless LAN consists of fixed hosts (or base stations) and mobile hosts. The fixed hosts are Pentium (133 MHz) based desktop machines running the Linux operating system (version 2.1.24).

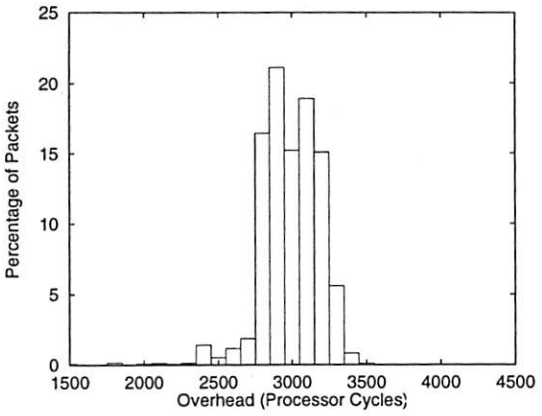


Figure 4: Overhead at the sender

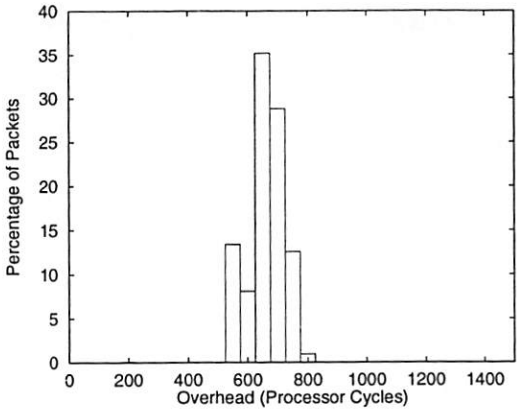


Figure 5: Overhead at receiver

The mobile hosts are Pentium (133 MHz) based laptops and also run Linux (version 2.1.24). The base stations as well as the mobile hosts use 2-Mbps WaveLAN technology for wireless communication. In addition, the base stations have a wired interface to a 10-Mbps Ethernet. To support mobility, the machines run the mobile-IP code developed at the State University of New York at Binghamton [11].

5.1 Overheads

The overheads introduced by the transformer tunnels depend on the adaptations requested. A complex encryption function will lead to reduced throughput. Tunnel overheads should thus be measured when the transformation function introduces a little extra overheads of its own. For this purpose, we have implemented the identity transformation function that leaves all packets unmodified. The same function is used to recover the packets at the mobile host. All packets are sent with the new protocol IP_XFORM, and the corresponding protocol handler is invoked at the mobile host. (In practice, such unmodified packets would be sent without the new protocol, and no processing would be required at the other end.)

Thus the overheads involved in the process are as follows.

- Sender: The tunnel changes the IP header, invokes the identity function, adds the metadata to the IP packet, and recomputes the IP checksum. It then sends the packet over the wireless interface.
- Receiver: The protocol handler for IP_XFORM removes the metadata, calls the identity function, restores the old protocol in the IP header, and recomputes the IP checksum. It then sends the packet back to the protocol stack.

To measure the overheads introduced by the transformer tunnels, we measured (using the internal Pentium counters) the processor cycles consumed by an identity transformation function. A transformer tunnel was established between a fixed host and a laptop. During the experiment, no other user processes were running either on the fixed host or on the laptop. The processor cycles were measured at both ends of the tunnel. The measurements were taken for ICMP (using *ping*) packets as well as UDP and TCP packets (using *ttcp*) from the base station to the mobile host. The overheads were also measured for various packet sizes. The results were same in all the cases. This is expected, because the operations performed by the tunnel and the identity function do not depend on the packet size or on the protocol being used. Figures 4 and 5 show the distribution of the overheads observed. At the base station, overheads are around 3100 processor cycles ($\approx 23\ \mu$s). Overhead for restoring the packet at the mobile host are around 700 processor cycles ($\approx 5\ \mu$s). In both the graphs shown above, a few (less than 5 in 3000) stray values have been discarded.

5.2 Buffering

	Avg loss % [St. Err]	Avg retransmissions % [St. Err]
No buffering	2.32 [0.36]	14.19 [0.54]
With buffering	0.63 [0.16]	6.91 [0.72]
% reduction	73	54

Table 2: Effect of buffering tunnel on handoffs

To evaluate the buffering tunnel, we used *raplayer* and *raserver* (from RealNetworks, Inc. [20]). For this experiment, we used two base stations, an audio server (another fixed machine), and a mobile host. Handoffs were forced every 10 seconds. The mobile host fetched audio files (total 1.2 MB). The buffer size used was 5 packets. Table 2 shows average percentage of packets lost (could not be played) due to handoffs and the percentage of packets that had to be resent by the server. The statistics were gathered from the server's access log files. The server was run in a mode where it logs all the information about lost packets.

Data type	Bytes saved (per KB) [St. Err]	Compression time (ms/KB) [St. Err]	Decompression time (ms/KB) [St. Err]
text	375 [0.789]	0.458 [0.001]	0.087 [0.000]
text (compressed)	0 [0]	0.303 [0.001]	0 [0]
image (GIF)	0 [0]	0.289 [0.001]	0 [0]
PostScript	469 [0.241]	0.386 [0.000]	0.090 [0.001]
PostScript (compressed)	0 [0]	0.290 [0.000]	0 [0]

Table 3: Compression

5.3 Reassembly

To evaluate the reassembly tunnel, we performed the following experiment. A realaudio server was placed on a fixed machine in our network. The mobile host accessed an audio file on the server. As before, we used *raplayer* to fetch this file. *raplayer's* statistics window was used to measure the number of packets that were delayed (and hence dropped). We also measured the total number of packets on the link. *raplayer* saw exactly the same number of packets because the original packets were regenerated when the reassembled packets reach the mobile host. The actual number of packets on the link was measured by snooping on the link (using *tcpdump*).

For the data stream in our experiment, we observed that the inter-arrival time for packets was close to 20 ms. Thus, we configured the tunnel with the delay parameter of 25 ms *i.e.* small packets (of size 256 bytes) that could be combined were delayed by at most 25 ms. Combined packets (each of size 512 bytes) were delivered as soon as possible. We observed that even with the reassembly tunnel, no packets were lost due to delay. In other words, the quality of the audio player was not affected by the reassembly transformation. At the same time, the number of packets sent over the link (and hence the bytes used by link-layer headers) were reduced by a *factor of two*.

5.4 Compression

We tested the compression function over WaveLAN. We measured the effect of compression on three data types: text, image and PostScript. In all the cases, we measured the number of bytes saved by compression, the compression overheads (measured in processor cycles, shown as time) and the decompression overheads. We also measured these parameters after compressing the original files with *gzip*. (The GIF files were not compressed as they are already in a compressed format.) Table 3 shows the results of the experiment. The numbers are averages over 10 sessions. During each session, 20 files were retrieved using *ncftp*. The results show that for compressible files (text and PostScript), the bytes saved due to compression more than compensate for the compression overheads. (On a 2-Mbps WaveLAN, assuming no transmission overheads, transferring one byte requires 3.8 μsec.)

6 Related Work

The problem of adapting to a changing environment has been studied extensively in the literature. Support from the base station has been used for adaptation at various layers of the protocol stack. I-TCP [3] and Snoop TCP [6] use base stations to improve TCP performance for mobile hosts. At the network layer, base stations have been used to improve the performance during handoffs [9]. Even at the link layer, use of base stations has been suggested for scheduling packet transmissions to reduce losses [8]. These mechanisms provide a fixed transformation function to solve specific problems introduced by mobility. Transformer tunnels suggested in this paper is not an alternative for above solutions. It merely provides a simple way of using such features whenever they are appropriate.

Some other mechanisms extend system functionality by interposing agents [22] between applications and the kernel. The University of Arizona's *x*-Kernel [19] provides support for composing protocols from simple elements. Protocol boosters [13] provide a way to insert such elements into the kernel "on-the-fly". Such protocol boosters are similar to the transformation functions described in this paper. Support for protocol boosters requires that every packet be inspected by the booster code to decide if boosting/deboosting is required (and a change in the kernel is also required for providing the booster support). The transformation functions described in this paper are used for dealing with specific link conditions rather than providing a generic support for modifying protocols. We use a simple and restricted mechanism to deal with transformations at the link layer. In our framework, only the packets to be transformed are sent to the tunnel, and only the packets that require inverse transformations are intercepted by the other end of the tunnel. All this is achieved without any changes to the kernel code.

There are application-layer adaptation techniques that allow greater flexibility. These techniques provide adaptations that are specific to some applications or application-layer protocols. In the Odyssey architecture [26], the granularity of data being sent over the network is decided based on the available resources. The Daedalus system [15] uses proxies to performs on-demand distillation of data. That includes converting color images to black-and-white images, converting PostScript to Rich Text Format (RTF), dropping

video frames, and so on. Another way of adapting to link conditions is by using "high-level proxies (HLP)" [37]. An HLP allows developing proxies similar to the Daedalus system.

There are several systems that provide mechanisms for safely adding code to the system for configuring protocols. For example, the SPIN operating system [14] allows dynamic configuration of protocols. The ANTS toolkit [36] allows even more flexibility by shipping the processing code along with the packet. Another approach is at language level where the PLAN (Programmable Language for Active Networks) [18] language is used for programs that are carried in the packets of a programmable network. The SwitchWare [30] project suggests use of "switchlets" for dynamically linking new functionalities with network elements.

7 Conclusions and Future Work

Today's networks are composed of links with diverse properties. In such networks, packet flow that is fine tuned (by selecting a proper packet size, transmission rate, encryption method, etc.) for some links may be inappropriate for other links. In this paper, we have shown that *transformer tunnels* provide an efficient mechanism for transforming the flow on various segments of a network, without affecting rest of the network. We have demonstrated the effectiveness of this technique by implementing it over our wireless network to improve the link utilization by compressing data and reassembling small packets, and to reduce losses during handoffs by buffering packets. We have also provided an API for easy development of transformation functions. This API has been used to implement the transformation functions described in this paper. We have implemented the tunnel manager as well that allows clients to control the transformations performed by transformation agents.

In the current implementation, the transformations to be performed depend on a packet's destination. We are investigating an extension to this approach where more restrictive filters can be specified. This can be useful for rerouting certain packets (such as rerouting all e-mail traffic to the home server). Moreover, packets can be selectively sent to a proxy server allowing easy development of client-proxy-server applications.

We also plan to combine transformer tunnels with a mechanism for exposing link conditions to higher layers of the protocol stack [32]. This will allow automatic reconfiguration of the tunnels in response to changes in the link conditions.

8 Acknowledgments

We would like to thank the anonymous reviewers for their valuable insight, detailed comments, and also for providing pointers to related work on protocol boosters. The reviews helped in improving the quality of the paper. We would also like to thank William Marcus for providing details on the work about protocol boosters.

References

[1] R. Atkinson. RFC1825: Security architecture for the Internet Protocol, August 1995. http://globecom.net/(nobg)/ietf/rfc/rfc1825.shtml.

[2] N. G. Badr. Cellular Digital Packet Data CDPD. In *Proceedings of the IEEE 14th Annual International Phoenix Conference*, pages 659–665, March 1995.

[3] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 136–143, Vancouver, Canada, May 1995.

[4] Hari Balakrishnan. Discussion on the end2end mailing list, 20 February 1996. ftp://ftp.isi.edu/end2end/end2end-interest-1996.mail.

[5] Hari Balakrishnan, Venkata Padmanabhan, and Randy Katz. The effect of asymmetry on TCP performance. In *Proceedings of the 3rd MOBICOM Conference*, pages 77–89, Budapest, Hungary, September 1997.

[6] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st MOBICOM Conference*, Berkeley, CA, November 1995.

[7] Bay Area Research Wireless Access Networks (BARWAN). http://http.cs.berkeley.edu/~randy/Daedalus/BARWAN/BARWAN_over.html.

[8] P. Bhagwat, P. Bhattacharya, A. Krishna, and S. K. Tripathi. Enhancing throughput over wireless LANs using channel state dependent packet scheduling. In *Proceedings of the INFOCOM*, March 1996.

[9] Ramón Cáceres and N. Padmanabhan. Fast and scalable handoffs for wireless internetworks. In *Proceedings of the 2nd MOBICOM Conference*, pages 56–66, November 1996.

[10] Mikael Degermark and Stephen Pink. Soft state header compression for wireless networks. In *Proceedings of the 2nd MOBICOM Conference*, pages 1–14, November 1996.

[11] Abhijit Dixit, Vipul Gupta, and Ben Lancki. Linux mobile IP: Implementation overview. http://anchor.cs.binghamton.edu/~mobileip/.

[12] Robert Durst, Gregory J. Miller, and Eric J. Travis. TCP extensions for space communication. In *Proceedings of the 2nd MOBICOM Conference*, pages 15–26, November 1996.

[13] D. C. Feldmeier, A. J. McAuley, and J. M. Smith. Protocol boosters. To appear in IEEE JSAC Special Issue on Protocol Architecture for the 21st Century, 1997.

[14] Marc E. Fiuczynski and Brian N. Bershad. An extensible protocol architecture for application-specific networking. In *1996 Winter USENIX Technical Conference*.

[15] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand distillation. In *Proceedings of the ASPLOS*, pages 160–170, October 1996.

[16] R. H. Frenkiel and Tomasz Imielinski. Infostations: The joy of "many-time many-where" communications. Technical Report 119, WINLAB, Rutgers University, April 1996.

[17] J. J. Garcia-Luna-Aceves et al. Wireless internet gateways (WINGS). In *Proceedings of the IEEE MILCOM*, Monteresy, California, November 1997.

[18] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. Submitted to PLDI, 1998.

[19] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[20] RealNetworks, Inc. Live and on-demand audio, video and animation for the Internet. http://www.real.com.

[21] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low speed serial links, 1990.

[22] M. B. Jones. Interposing agents: Transparently interposing code at the system interface. In *Proceedings of the 14th SOSP*, pages 80–93, Asheville, NC, 1993.

[23] R. Frank Quick Jr. and Kumar Balachandran. An overview of the cellular digital packet data (CDPD) system. In *Proceedings of the Fourth International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Yokohama, Japan, September 1993.

[24] Jacob R. Lorch and Alan Jay Smith. Software strategies for portable computer energy management. To appear in IEEE Personal Communications.

[25] Metricom ricochet modem. http://www.metricom.com/ricochet/.

[26] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th SOSP*, 1997.

[27] Markus Franz Xaver Johannes Oberhumer. miniLZO – mini version of the LZO real-time data compression library. http://www.infosys.tuwien.ac.at/Staff/lux/marco/lzo.html.

[28] A. Sacham, R. Monsour, R. Pereiera, and M. Thomas. IP payload compression protocol (IPComp), October 1997. Internet draft, Work in progress.

[29] V. Schryver. RFC1977: PPP BSD compression protocol, August 1996. http://globecom.net/(nobg)/ietf/rfc/rfc1977.shtml.

[30] J. M. Smith, D. J. Farber, C. A. Gunter, Scott Nettles, D. C. Feldmeier, and W. D. Sincoskie. Switchware: Accelerating network evolution (white paper), 1996. http://www.cis.upenn.edu/~jms/white-paper.ps.

[31] Mark Stemm, Paul Gauthier, Daishi Harada, and Randy Katz. Reducing power consumption of network interfaces in hand-held devices. In *Proceedings of the 3rd International Workshop on Mobile Multimedia Communications (MoMuc-3)*, pages 130–142, 1996.

[32] Pradeep Sudame and B. R. Badrinath. On providing support for protocol adaptation in mobile wireless networks. Technical Report 333, Department of Computer Science, Rutgers University, July 1997. http://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-333.ps.Z.

[33] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, second edition, 1993. Section on cryptography.

[34] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. Minden. A survey of active network research. In *IEEE Communications*, January 1997.

[35] Wavelan. http://www.wavelan.com/.

[36] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. Submitted to IEEE OPENARCH, 1998.

[37] Bruce Zenel and Dan Duchamp. A general purpose proxy filtering mechanism applied to the mobile environment. In *Proceedings of the 3rd MOBICOM Conference*, pages 248–259, Budapest, Hungary, September 1997.

The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator

Marc E. Fiuczynski Vincent K. Lam Brian N. Bershad
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

Abstract

IPv6 is a new version of the internetworking protocol designed to address the scalability and service shortcomings of the current standard, IPv4. Unfortunately, IPv4 and IPv6 are not directly compatible, so programs and systems designed to one standard can not communicate with those designed to the other. IPv4 systems, however, are ubiquitous and are not about to go away "over night" as the IPv6 systems are rolled in. Consequently, it is necessary to develop smooth transition mechanisms that enable applications to continue working while the network is being upgraded. In this paper we present the design and implementation of a transparent transition service that translates packet headers as they cross between IPv4 and IPv6 networks. While several such transition mechanisms have been proposed, ours is the first actual implementation. As a result, we are able to demonstrate and measure a working system, and report on the complexities involved in building and deploying such a system.

1 Introduction

The current internetworking protocol, IPv4 [11], eventually will be unable to adequately support additional nodes or the requirements of new applications. IPv6 is a new network protocol that features improved scalability and routing, security, ease-of-configuration, and higher performance compared to IPv4. Unfortunately, IPv6 is *incompatible* with IPv4 and to use the new protocol will require changes to the software in every networked device. IPv4 systems, however, are ubiquitous and are not about to go away "over night" as the IPv6 systems are rolled in. Consequently, it is necessary to develop transition mechanisms that enable applications to continue working while the hosts and networks are being upgraded. One suggested strategy is to translate IP headers as they cross between IPv4 and IPv6 networks [3]. The requirement of header translation is to remain transparent to applications and the network.

In this paper we present two variations of IPv6/IPv4 translators that address these difficulties. The first variation uses *special* IPv6 addresses, as proposed in [4], to easily translate packets transparently for all applications. Unfortunately, these special IPv6 addresses also require IPv6 routers to contain special routes to them, which is considered to be a bad idea because it creates more state for the router to maintain [4]. The second variation maintains an explicit mapping between IPv4 and IPv6 addresses, and is therefore able to use standard IPv6 addresses that do not require any special treatment by IPv6 routers. Its drawback is that IP-addresses embedded in some applications' data stream, such as FTP, must be updated as well for the translation to be completely transparent.

We have built an IPv6/IPv4 network address and protocol translator as a device driver running in the Windows NT operating system [15]. Our test environment consists of the translator as a gateway between IPv6 and IPv4 hosts connected to separate Ethernet segments, and it incurs little performance overhead. Between a pair of IPv6 and IPv4 nodes communicating via the translator, we have measured TCP bandwidth of 7210 Kbytes/second and roundtrip packet latencies of 424 microseconds over 100Mbit/second Ethernet links.

1.1 Motivation

Our efforts began with an implementation of the IPv6 protocol for the SPIN [13] extensible operating system, which enables the rapid prototyping of kernel extensions. After completing the initial IPv6 implementation we connected our system to the 6Bone [12]. We were interested in accessing services using IPv6, but quickly discovered that there were only a few hosts (roughly 250) accessible via the 6Bone with even fewer IPv6 native services to talk to. Thus, we decided to build an IPv6/IPv4 translator to enable IPv6 systems to access the IPv4 systems and services, and vice versa. There are two main scenarios where network address and protocol translation are applicable:

- An IPv6 site communicating with IPv4 nodes. For example, a completely new network with new devices that all support IPv6 may occasionally need to communicate with some IPv4 nodes out on the Internet.
- An IPv4 site communicating with IPv6 nodes. For example, upgrading an IPv4 site to IPv6 on a node-by-node basis requires that critical services, such as

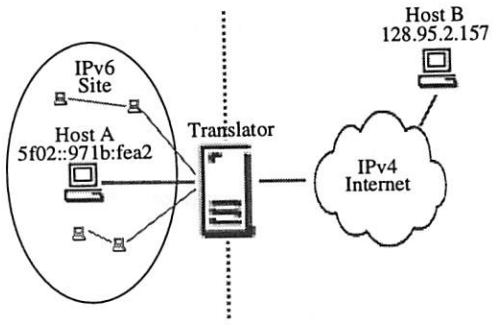


Figure 1. Translator for an IPv6 site.

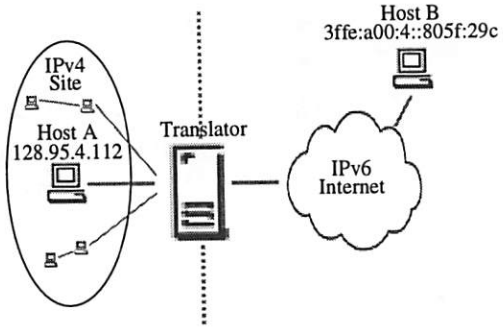


Figure 2. Translator for an IPv4 site.

web, file, and print services are accessible from both IPv6 and IPv4 nodes.

The rest of this paper describes the design and implementation of the IPv6/IPv4 translator and is organized as follows. In Section 2 we describe network address and protocol translation. In Section 3 we present the applications and benchmarks used to test the translator. In Section 4 we discuss possible solutions for some unresolved issues. In Section 5 we survey related work regarding network address and protocol translation. Finally, in Section 6 we conclude.

2 Network Address and Protocol Translation

The address and protocol translation presented in this section enables both the communication between nodes in an IPv4 site with nodes in the IPv6 network, and between nodes in an IPv6 site with nodes in an IPv4 nodes. Figures 1 and 2 illustrate these scenarios, and the following paragraphs describe them in more detail.

Figure 1 illustrates a translator for an IPv6 site communicating with nodes in an IPv4 network. The internal routing of the IPv6 site must be configured such that packets intended for IPv4 nodes route to the translator. Hosts in the IPv6 site send packets to nodes in the IPv4 network using IPv6 addresses that map to individual IPv4 hosts. For this scenario, a design presented in [4] proposes that IPv6 nodes use an *IPv4-compatible IPv6* address as their own address and an *IPv4-mapped IPv6* address when communicating with IPv4-only nodes. An IPv4-compatible IPv6 address holds an IPv4 address in the low-order 32-bits, with a unique high-order 96-bit prefix of `0:0:0:0:0:0` (all zero bits), and always identifies an IPv6/IPv4 or IPv6-only node; they never identify an IPv4-only node. Similarly, an IPv4-mapped IPv6 address identifies an IPv4-only node and its high-order 96-bits bear the prefix `0:0:0:0:0:FFFF`. The address of any IPv4-only node may be mapped into the IPv6 address space by prefixing `0:0:0:0:0:FFFF` to its IPv4 address. The benefit of this approach is that the translator can be stateless. However, regardless of the 96-bit IPv6 prefix that is used to map between the IPv4 and IPv6 address domains it still remains necessary to identify a host in the IPv6 site with an unique IPv4 address. That is, in Figure 1, for Host B to communicate with Host A requires an IPv4 address that can be routed through the IPv4 Internet. To overcome this limitation a stateful translator could multiplex several IPv6 hosts onto a single, globally unique IPv4 address using the TCP/UDP port translation technique described in [2].

Figure 2 illustrates a translator for an IPv4 site communicating with nodes in an IPv6 network. Hosts in the IPv4 site send packets to nodes in the IPv6 network using IPv4 destination addresses assigned by the translator that map to individual IPv6 hosts. For this to work, the internal routing of the IPv4 site must contain routes to the translator for packets with the destination field using one of these IPv4 addresses. The translator, upon receiving such packets, will do the IPv4-to-IPv6 translation and forward the packet to the IPv6 network. In contrast to the above scenario, the translator can use unique IPv6 addresses to refer to nodes in the IPv4 site in order to do IPv6-to-IPv4 translation for packets it receives from the IPv6 network. These IPv6 addresses may come from a pool that is dynamically assigned to the set of IPv4 hosts communicating with IPv6 hosts. A better approach is to assign unique and routable IPv6 addresses to all nodes in the IPv4 site and to register them with DNS. This should be easily possible given that the IPv6 address space is sufficiently large, and also has the benefit that arbitrary hosts in the IPv6 Internet can easily lookup and initiate sessions with nodes in the IPv4 site via the translator.

In summary, the subtle difference between these two scenarios is that the former involves mapping a pool of *global* IPv4 addresses referring to IPv6 addresses, whereas the latter can leverage site *private* IPv4 addresses to refer to IPv6 addresses. Global IPv4 addresses will be scarce and mechanisms are required to dynamically assign a pool of these IPv4 addresses on a temporary basis to IPv6 nodes so that they can

communicate with IPv4 nodes. On the other hand, there is a large pool of roughly 17 million site private IPv4 addresses defined by [14], which can be used by the translator to map to IPv6 addresses. Our translator is designed to support all of the scenarios just described.
To enable communication between an IPv4 and IPv6 node, a translator needs to do both address and protocol translation. Protocol translation involves mapping most of the fields illustrated in Figure 3 from one version of IP to the other. Address translation involves converting addresses for packets crossing the protocol boundary.

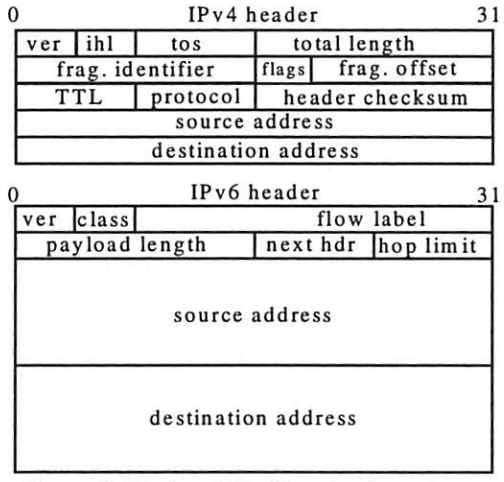


Figure 3. IPv4 and IPv6 header format.

The following two subsections describe the address and protocol translation process in further detail.

2.1 Address Translation

Address translation is trivial when using IPv4-mapped and IPv4-compatible IPv6 addresses. For the IPv6-to-IPv4 direction the translator simply extracts the lower 32-bits of an IPv6 address to obtain an IPv4 address. For the opposite direction the translator sets the lower 32-bits of the IPv6 source/destination addresses to the IPv4 source/destination addresses, and sets the upper 96-bits of the IPv4 source and destination addresses to the IPv4-mapped and IPv4-compatible prefix, respectively. However, it is considered to be a very bad idea to use IPv4-mapped address as it has the drawback of requiring IPv6 routers to contain routes to IPv4-mapped addresses [4]. The alternative is to use IPv6-only addresses to refer to IPv4 nodes, which requires the translator to maintain an explicit mapping between IPv4 and IPv6 addresses.
For clarity, we introduce an IPxNODEy notation to disambiguate among the types of addresses used in the translation process. Table 1 defines the four types of addresses in terms of this notation. The first two rows define the addresses that are native to the IPv4 and IPv6 nodes. The last two rows define address aliases, which

IpxNODEy	Definition
IP4NODE4	v4 address of a 4 node
IP6NODE6	v6 address of a v6 node
IP6NODE4	v6 address referring to a v4 node
IP4NODE6	v4 address referring to a v6 node

Table 1. IP address definition.

are assigned by the translator, used to translate between the IPv4 and IPv6 address domains.
As an example of using this IPxNODEy notation consider the following scenario: an arbitrary IPv6-only host wishes to communicate with our IPv4-only web server via the translator. For an IPv6 host to communicate with our IPv4 web server requires an IPv6 address that is an alias (IP6NODE4) address for the web server's native IPv4 host (IP4NODE4) address. Similarly, for the web server to reply to the IPv6 host requires an IPv4 address that is an alias (IP4NODE6) address for the IPv6 host's native (IP6NODE6) address. That is, the translator maps the IP6NODE4 address to the IP4NODE4 address of the web server, and the IP4NODE6 address to the IP6NODE6 address of the IPv6 host.
The translation of addresses has three phases: address binding, address lookup and translation, and address unbinding, which we describe in the following subsections.

2.1.1 Address Binding

Address binding is the phase where an IPv4 address is associated with an IPv6 address and vice versa. The translator maintains key-to-value tuples, listed in Table 2, to map between IPv4 and IPv6 addresses.

Key-to-Value	Definition
IP6NODE4-to-IP4NODE4	v6 addresses mapped to v4 node addresses
IP4NODE6-to-IP6NODE6	v4 addresses mapped to v6 node addresses

Table 2. Mappings between IPv4 and IPv6 addresses used by translation process.

For addresses that are statically mapped, the binding happens when the translator is initialized. If the translator is configured to use IPv4 mapped/compatible IPv6 addresses then all the bindings are implicitly static as they are defined by these special IPv6 addresses. Other static mappings could be setup between arbitrary IPv4 and IPv6 addresses. For example, the binding of addresses for an IPv4 node to an IPv6 node could be done statically by a network manager when assigning IPv6 addresses to existing nodes in the IPv4 site. That is, IP6NODE4-to-IP4NODE4 are static mappings of IPv6 addresses assigned to IPv4 hosts. Otherwise, the

binding between addresses needs to happen dynamically.
IPv6 addresses are larger than IPv4 addresses and it is not possible to create a one-to-one IP4NODE6-to-IP6NODE6 binding. Consequently, it will be necessary to reuse IP4NODE6 addresses to bind them to other IP6NODE6 addresses. In Section 2.1.3 we discuss this issue in more detail.

2.1.2 Address Lookup and Translation

Once a binding is established it can be used for address lookup and translation. The example in Figure 4 illustrates the translation using the IPxNODEy notation defined earlier. When the IPv4 node sends a packet to

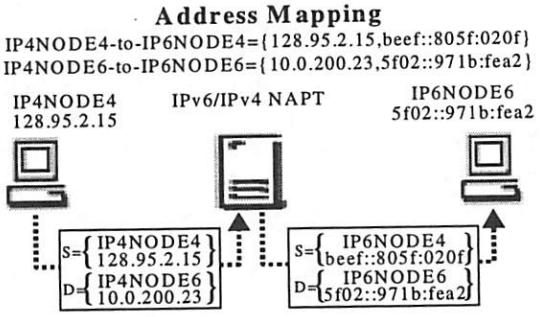


Figure 4. Basic address translation operation.

the IPv6 node it is routed through the translator. The translator receives the packet, translates the 128.95.2.15 to beef::805f:020f source address using the IP4NODE4-to-IP6NODE4 mapping, and translates the 10.95.2.23 to 5f02::971b:fea2 destination address using the IP4NODE6-to-IP6NODE6 mapping. Likewise, IP packets on the return path go through a reverse address translation.
Notice that this requires no changes to hosts or routers. As far as the IPv4 host is concerned, IP4NODE6=10.0.200.23 is the address used by the IPv6 hosts. Conversely, the IPv6 host believes that IP6NODE4=beef::805f:020f is the address used by the IPv4 hosts. The address translation is transparent to both hosts.

2.1.3 Address Unbinding

Address unbinding is the phase when the association between an IPv4 and IPv6 address is broken. We expect the number of bindings of the IP6NODE4-to-IP4NODE4 mapping to remain fairly constant during the day-by-day operation of the translator; new bindings are only necessary when adding new hosts to the site.
On the other hand, the number of bindings of the IP4NODE6-to-IP6NODE6 mapping are more dynamic and depend on the number of connections established to different hosts in the network. The number of reserved IP4NODE6 addresses used by the translator limits the number of bindings possible for the IP4NODE6-to-IP6NODE6 mappings.
For the scenario where the translator is providing service for an IPv6 site (as illustrated in Figure 1), the IP4NODE6 addresses are a small number of unique IPv4 addresses. It is crucial for the translator to detect when an IP4NODE6 address can be reused in order to create new bindings; otherwise, new sessions may be refused if there are no IP4NODE6 addresses available.
For the scenario where a translator is providing service to an IPv4 site (as illustrated in Figure 2), the IP4NODE6 addresses may come from a relatively large pool of private network addresses (as mentioned earlier, there are roughly 17 million of such addresses available). Here the concern is to safely remove unused bindings to ensure that the mapping table does not require too much memory and that address lookup performance does not deteriorate. Removing a binding too early should never occur, as it would effectively terminate any ongoing communication that relied on the binding.

2.2 Protocol Translation

Protocol translation consists of a simple mapping between the two IP protocols, with some special rules for handling fragments and path MTU discovery. The basic operation is to remove the original IP header and replace it with a new header from the other IP version. The rest of this section provides a high-level overview of the protocol translation process and the issues involved. In the Appendix of this paper we present the details of protocol translation between IPv4/IPv6 and ICMPv4/ICMPv6 headers.

2.2.1 IP Translation

The IPv6 and IPv4 headers have some similarity, but there are a number of fields that are either missing or have different sizes or meaning. The translator either directly copies, translates, ignores, or sets fields in the IP header to a default value when translating from one version of IP to the other. Figure 5 illustrates the actions taken by the translator for each header field.
Many of the fields require a simple adjustment. The IPv4 *checksum* field is computed when translating from IPv6-to-IPv4, and ignored when translating from IPv4-to-IPv6. The IPv4 *total-length* field includes the IPv4 header size whereas the IPv6 *payload-length* field does not. The translation needs to account for this difference. The *hop-limit/time-to-live* fields are copied and decreased by one. Finally, the *protocol* field can be directly copied from one version of IP to the other, with ICMPv4 and ICMPv6 protocol numbers being the only exception.

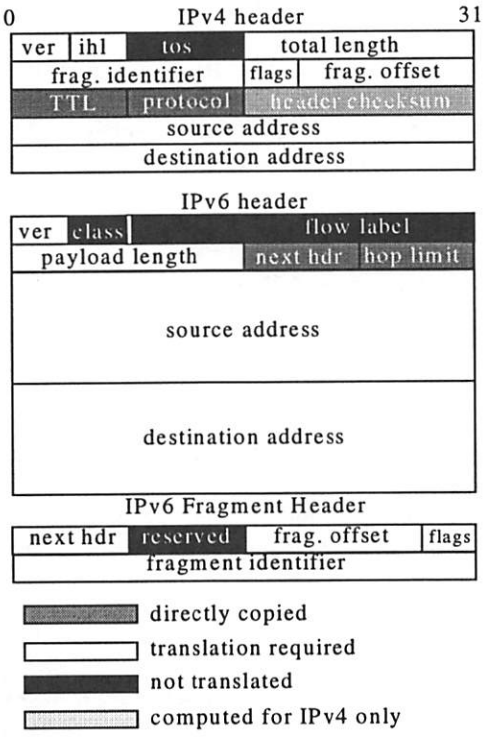


Figure 5. This Figure illustrates which fields of the IPv6/IPv4 header are directly copied, require translation, or are ignored. In contrast to IPv4, the IPv6 header does not have explicit fields to support fragmentation; it uses a separate Fragment header for this information.

With the exception of the IPv6 Fragment header, all other IPv6 extension headers and IPv4 options are silently ignored by the translator. The IPv4 *type-of-service* and IPv6 *traffic-class* and *flow-label* fields are also ignored by the translator, as there does not exist a semantic mapping between them (specifically, the use of the IPv6 flow-label field has not been specified yet). We discuss this loss of information in Section 4.1 further.

When the translator receives a fragmented packet, the translation is straightforward since there is a direct mapping between the IPv4 and IPv6 fragmentation fields. The only caveat is the size difference of the fragment identifier field between the two protocols. In IPv6, this field is 32-bits wide and twice as large as its IPv4 counterpart. To account for this, we currently just copy the lower 16 bits of the IPv6 fragmentation identifier when translating from IPv6 to IPv4.

Whenever the translator encounters a non-fragment IPv4 packet with the *Don't Fragment* flag set to false (i.e., fragmentation is allowed for that packet), it notes that by adding an IPv6 Fragment header and copying the IPv4 fragmentation fields to it, which indicates the following:

1. The sender allows fragmentation and that the fragmentation information is carried end-to-end to ensure that packets are correctly reassembled.
2. The sender is not using path MTU discovery and the *Don't Fragment* bit must be set to false should the packet be translated back to IPv4.

The translation from IPv4 to IPv6 increases the packet size by at least 20 bytes due to the header length difference between the two protocols (28 bytes if it needs to add a Fragment header). If the *Don't Fragment* flag is set to true and the resulting packet is greater than the next-hop MTU, then the translator will return an ICMP error message (Packet Too Big). Otherwise, the translator will fragment the resulting packet into next-hop MTU-sized packets. Note that this fragmentation results in an inefficient packet stream in the case where the IPv4 host is sending MTU-sized packets (e.g., a network file system, such as NFS). For this situation, we are experimenting with returning ICMPv4 "Packet Too Big" error message to the IPv4 host that contains a next-hop MTU that accounts for the size difference in the IP header size, giving the host the opportunity to re-adjust its path MTU value. If the host continues to send large packets (i.e., it does not support path MTU discovery), then the translator will stop sending the ICMP error message and continue fragmenting the packet.

2.2.2 ICMP Translation

The translator silently drops single hop ICMP messages as well as ICMP messages with unknown Type fields. For the remaining ICMP messages the header format is nearly identical for ICMPv4 and ICMPv6. The only exception is the ICMP Parameter Problem message, which an 8-bit pointer value in ICMPv4 and a 32-bit pointer value in ICMPv6. The following ICMP messages and errors have a counterpart in each version: Echo Request, Echo Reply, Time Exceeded, Destination Unreachable, Packet Too Big, and Parameter Problem. For most cases there is a simple translation of the ICMP Type and Code fields. When a Packet Too Big error message reaches the translator, it needs to adjust the Maximum Transmission Unit (MTU) field during the translation to account for the

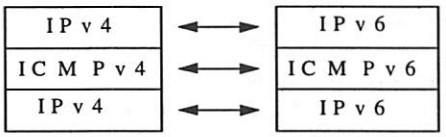


Figure 6. ICMP error messages include the IP header of the error causing packet, which must be translated as well.

difference between IPv4 and IPv6 header sizes. Also, for a Parameter Problem error message the Pointer field needs to be adjusted to point to the corresponding field in the error causing IP header.

ICMP error messages contain as much of the error invoking packet's IP header and data as can fit, and needs to be translated just like a normal IP header that delivered the message. That is, it requires a recursive translation of the IP packet contained in the ICMP error message, as illustrated in Figure 6. The caveat is that the translation of the IP header is likely to change the length of the datagram, in which case the IPv6 Payload-length and IPv4 Total-length fields need to be adjusted as well. Finally, the translator silently drops all IGMP messages.

2.2.3 Adjusting Checksum Values

Several higher-layer protocols (e.g., TCP, UDP) compute their checksum values on a pseudo-header that consists of fields from the IP header. The checksum value needs to be adjusted with the difference between the original IP addresses and the translated IP addresses.

The checksum adjustment for ICMP is slightly more complex. ICMPv6 uses a pseudo-header checksum similar to UDP and TCP, whereas ICMPv4 does not. For ICMP Echo and Echo Request informational messages we calculate the incremental checksum adjustment, as only the Type value changes. When translating from ICMPv6 to ICMPv4 we need to subtract the pseudo-header checksum. Conversely, when translation from ICMPv4 to ICMPv6 we need to add the pseudo-header checksum. Note that these informational messages may be fragmented either by the sending host or intermediate routers if their size exceeds the path MTU. For this case, the translator cannot calculate the correct checksum value for ICMP Echo and Echo Request messages, because it does not know the total size of the packet, which it requires to add/subtract the pseudo-header checksum value when translating between the ICMP versions. Finally, since ICMP error messages are never fragmented, our approach is to recalculate the checksum value from scratch rather than incrementally, because most of the ICMP header and data values have changed.

3 Implementation

In this section we present basic performance measurements and describe a set of applications that we have used to verify whether the translator works for real applications. Our experimental setup consists of IPv6 and IPv4 machines connected to separate, private Ethernet segments. The translator is equipped with two Ethernet cards and acts as a gateway between the IPv6 and IPv4 Ethernet segments. All machines in our setup are Intel PCs equipped with a 200Mhz Pentium Pro processor, 64MB of RAM, and 3COM 3c905 fast Ethernet cards. We use both Linux (2.1.95) and Windows NT 4.0 as our IPv6 test machines. For Windows NT we use Microsoft Research's publicly released IPv6 stack [16]. The translator is implemented as a Windows NT device driver and roughly consists of 2000 lines of C code. It uses the IPv4 and IPv6 stacks in Windows NT to send IP packets.

3.1 Latency and Bandwidth

To evaluate the performance of the translator we used the `ttcp` tool to measure bandwidth and `ping` to measure latency between a pair of IPv6 and IPv4 hosts. We compare the packet forwarding performance of the IPv6/IPv4 translator with NT's built-in IPv4 forwarding support.

We measured the roundtrip latency of `ping` packets ranging in size from 64 bytes to 1440 bytes on 100Mbps Ethernet links. In Table 3, the columns labeled *v4-v4* and *v6-v6* show the latency between two machines communicating directly using the same protocol. The columns labeled *FWD* and *NAPT* show the roundtrip latency going through NT's forwarder and our translator, respectively. The translator is on average about 30 microseconds slower compared to the forwarder.

Msg. size in bytes	v4-v4	v6-v6	FWD	NAPT
64	246	244	397	424
128	262	261	448	463
256	297	295	508	540
512	364	360	630	658
1024	487	482	871	918
1440	603	596	1059	1104

Table 3. Roundtrip latency of PING packets measured in microseconds.

Table 4 shows the bandwidth of sending 64 Mbytes using TCP for both 10Mbps and 100Mbps Ethernet. Note that for 10Mbps Ethernet the overhead of the translator and the forwarder are essentially unnoticeable. However, the bandwidth for the forwarder and the translator on fast Ethernet is much lower compared to two machines communicating directly using either IPv4 or IPv6. Using NT's performance monitor we noticed that processor utilization reaches nearly 100% on our forwarder/translator machine when running the ttcp

Link Speed	v4-v4	v6-v6	FWD	NAPT
Ethernet	1095	1092	1093	1089
Fast Ether	11003	9076	8005	7210

Table 4. TCP bandwidth measured in Kbytes/second.

bandwidth benchmark over fast Ethernet. The reason for the high CPU utilization is NT's packet receive architecture, which assumes the device driver owns the packet buffer rather than passing buffer ownership to the module receiving the packet (as is the case in most UNIX systems). Consequently, we believe that bandwidth through the translator and the forwarder are CPU limited, as they incur significant overhead due to NT's packet receive architecture; they must allocate buffer space for the IP packet's payload and copy the data in its entirety before being able to forward it. Additionally, note that the bandwidth through the translator is 10% slower compared to the forwarder. We attribute this performance degradation partly to the IPv6 prototype from Microsoft Research, which is roughly 1.9Mbytes/second slower than the production IPv4 stack shipped with Windows NT. We expect the end-to-end TCP bandwidth to improve as the IPv6 implementation for Windows NT matures.

We are pleased with the current latency and bandwidth measurements, as they indicate that translation does not inherently have a significant impact on performance.

3.2 Applications

The goal of the translator is to transparently work for "real world" applications, and we used a representative set of programs that exercise the TCP, UDP, and ICMP protocols via the translator. Our test applications consist of an IPv6 version of an Apache web-server, `ttcp`, `finger`, `telnet`, `ping`, `traceroute`, and `ftp`.

We knew from our experiments with `ttcp` that the TCP protocol translation works, but wanted to verify this with common TCP applications. We were able to use `telnet` and `finger` to connect between IPv6 and IPv4 hosts through the translator. Additionally, a web browser on an IPv4 host retrieving documents from an IPv6 Apache web-server was equally successful.

The `ping` program uses ICMP messages to determine whether a particular host is alive. We also used `ping` to measure basic roundtrip latency between hosts.

The `traceroute` program tracks the flow of a packet from router to router. When tracking routes from an IPv6 node through the translator along an IPv4 network, the addresses of the IPv4 routers are translated into IPv4-mapped IPv6 addresses. For the other direction, the translator establishes bindings, described in Section 2.1.1, for the IPv6 router addresses to private network addresses.

Although `ping` and `traceroute` use ICMP, they do not adequately test whether the recursive ICMP translation, described in Section 2.2.2, was working properly. Table 5 lists how we caused various ICMP error messages to verify their correct translation.

Finally, we tested `ftp`, which is an application that embeds an ASCII IP address and sends it to its peer.

ICMP Error Message	Error causing action
Destination unreachable	UDP packet to unreachable port
Packet Too Big	packet exceeding path MTU size
Time Exceeded	single incomplete IP fragment
Parameter Problem	packet with invalid field

Table 5. Error causing actions to verify ICMP translation.

For it to work correctly via the translator, the IPv6 implementation of the ftp client needs to detect whether the connection is with an IPv6 or IPv4 version of the ftp daemon. When communicating with an IPv4 ftp daemon it needs to use as an ASCII IP address of its host's IPv4-compatible IPv6 address instead of the host's native IPv6 address. Conversely, when an IPv4 ftp client contacts an IPv6 ftp daemon, the daemon must treat the ASCII IP address as an IPv4-mapped IPv6 address. With this approach it is not necessary for the translator to update the ASCII IP address.

4 Discussion

The previous section illustrated that the basic translation between the two IP protocols is possible for real applications. In this section, we discuss some unresolved issues regarding loss of information, applications with IP address content, and how IPv6 hosts resolve to IPv6 addresses referring to IPv4 hosts (i.e., IP6NODE4 addresses) and vice versa (i.e., IP4NODE6 addresses). Finally, we discuss an integrated translator approach that addresses the host lookup problem and address-unbinding problem mentioned in Section 2.1.3.

4.1 Loss of Information

Although a basic mapping exists between the two IP protocols there are certain fields, options, and extensions that cannot be translated. The result is a loss of information that may have some impact on applications. For example, IPv4 *type-of-service* values cannot be equivalently expressed in an IPv6 context where quality of service for a packet is marked by two fields, *traffic-class* and *flow-label*, as they differ in their currently specified semantics. Another example is the use of extension headers by IPv6. These headers can be of arbitrary length and can encapsulate options greater than the IPv4 limit of 40 bytes. Further, the IPv6 specification defines extensions for features such as Authentication, Encapsulation, and Extended Routing that are a superset of the IPv4 feature domain. Thus, it is not possible for fully transparent header translation to

occur without loss of information in cases where the disjoint functionality is exploited. Our current approach is to ignore all of these features during the translation process and observe the impact on applications. So far our experience is that applications generally rely on basic IP features and do not use the extended fields of the IP header.

4.2 Applications with IP Address Content

Some applications embed their IP addresses in the packet payload, above Layer 3. This is the case for a number of applications, including certain File Transfer Protocol (FTP) programs, and the Windows Internet Name Service (WINS) registration process of Windows 95 and Windows NT. Unless the translator parses every packet all the way to the application level, it has no way of translating embedded IP addresses, which can lead to application failures. Our implementation does not do any application-level IP address translation, but as described in Section 3.2, this is not an issue with new IPv6 applications that are IPv4-aware, like FTP. We hope that a similar solution can be used with IPv6 versions of all legacy applications that embed IP address content. If that's not possible, then the translator will need to be complemented with application level gateways to expand the list of supported applications [7].

4.3 Hostname Lookup

Before a host can initiate a session with another host it has to lookup its address. This is generally done using host tables or DNS. The problem when using a translator is that the lookup needs to resolve to an address alias that refers to the actual host. For the case where the translator enables nodes in an IPv4 site to communicate with nodes in the IPv6 network it is reasonable to assume that each IPv4 node has assigned to it a unique IPv6 address. Thus, arbitrary IPv6 nodes can lookup its address and initiate a session. However, the converse of an IPv4 looking up an IPv6 host is more difficult, as the IPv4 node needs to obtain the address alias from the translator that refers to the IPv6.

There are several approaches that can be taken to translate an IPv6 DNS record to an IPv4 DNS record. First, the resolver library of the IPv4 nodes could be modified to request the alias from the translator when encountering IPv6 DNS records. Second, the site's DNS servers could be modified to request a temporary address from the translator on behalf of its IPv4 clients when encountering an IPv6 DNS record. Finally, an approach proposed in [7] suggests that the translator recognize DNS request and response packets and translates them transparently.

The implications that IPv6/IPv4 translation has to DNS are beyond the scope of this paper, but need to be addressed for translation to be completely transparent.

4.4 The Integrated Approach

Our experience with a network-based translator revealed that for IPv6/IPv4 translation to be completely transparent requires varying degrees of integration with other services. As mentioned in the previous subsection, some level of cooperation is necessary between DNS and the translator to bind IPv4 addresses to IPv6 addresses, and vice versa. Also, the translator currently uses ad-hoc methods to detect when it can safely remove bindings. Our strategy is to integrate the translator functionality directly into an IPv6/IPv4 host operating system. There are several benefits of the integrated approach:

- *Failure isolation.* The integrated translator only serves the host that it is running on and its failure will not affect other hosts.
- *Scalability.* The integrated translator needs to scale only with the number of network applications running on the host, rather than the sum of network applications running in the site served by a network based translator.
- *Safe reclamation of address bindings.* The integrated translator is aware when an application terminates a TCP/UDP network connection and can safely unbind the address.

Finally, and most noteworthy, the integrated approach enables the illusion of an IPv6-only node, as packets stemming from legacy IPv4 applications may be translated to IPv6 before they leave the machine.

5 Related Work

In principle the function of IPv6/IPv4 address translation is similar to an IPv4 Network Address Translator (NAT) [2], which converts private internal addresses to globally unique addresses that are passed to the Internet backbone and vice versa. The IPv4 NAT has the following limitations. First, it is stateful in order to map between the globally unique and private internal addresses; thus the NAT is a single point of failure. Second, applications with IP-address content require special translation that may be difficult (such as updating ASCII IP strings and maintaining TCP sequence numbers on the fly), or may be impossible when the application data stream is encrypted or signed. Any stateful translator shares these limitations. Nevertheless, despite these limitations NATs seem to be widely used.

A proposal called "Network Address Translation – Protocol Translation" (NAT-PT) [7] presents a stateful IPv6/IPv4 translator design. It also describes how to incorporate IPv4 NAT style UDP/TCP port number

translation. With exception of the port number translation this is similar to the stateful component of our design.

A proposal called "Stateless IP/ICMP Translation" (SIIT) [4] avoids the need for address translation, thereby overcoming the limitations of IPv4 NAT. First, it does not maintain state, and is therefore resilient to network failure. Moreover, multiple stateless translators may be used to scale with larger sites. Second, the use of IPv4-mapped and IPv4-compatible addresses allows it to avoid translating IP addresses embedded in the application's data stream. However, this approach will only work if the IPv6 socket API treats mapped/compatible addresses exactly as IPv4 addresses. For example, as is the case for some FTP programs, mapped/compatible IPv6 addresses need to be printed as IPv4 ASCII strings. The drawback of the SIIT design is that IPv6 routers need to contain routes to IPv4-mapped addresses. This drawback seems acceptable when the translator serves an IPv6 site with access to the IPv4 Internet (e.g., the scenario shown in Figure 1). However, for the case where the translator serves an IPv4 site with access to the IPv6 Internet (e.g., the scenario shown in Figure 2) the use of IPv4-mapped/compatible IPv6 address is unreasonable, as it counteracts one of the significant benefits of IPv6: shrinking backbone routing tables.

Finally, a proposal called "Assignment of IPv4 Global Addresses to IPv6 Hosts" (AIIH) [5] enables dual-stack IPv6/IPv4 nodes to temporarily acquire a global IPv4 address to communicate with other IPv4-only nodes. This approach may be the initial stepping stone to allow sites to configure a large set of IPv6 hosts without having to statically assign each host a globally unique IPv4 address.

Both the SIIT and AIIH designs focus on providing interoperability between an IPv6 site and the IPv4 Internet, whereas stateful translation (e.g., NAT-PT) enables an IPv4 site to communicate with the emerging IPv6 Internet.

While several translator designs have been proposed [4][7], ours is the first actual implementation. Our translator implementation is based on the address translation techniques described in Section 2.1, which are general enough to support both stateful and stateless translation.

6 Conclusion

We have described the design and implementation of an IPv6/IPv4 network address and protocol translator, and briefly compared pros and cons of stateless vs. stateful translation. To this date there are three proposals [4][5][7] submitted to the IETF NGTRANS working group to support the interoperability between IPv6 and IPv4-only nodes. Our work subsumes both the stateless SIIT design described in [4] and the stateful design described in [7]. Despite the limitations of translation (e.g., loss of information) we believe that a translator can adequately fulfill the role of a short-term transition aid from IPv4 to IPv6, since it supports the majority of Internet traffic (HTTP, FTP, sendmail).

Based on our experience we conclude that an IPv6/IPv4 network address and protocol translator is complementary to the AIIH [5] approach in transitioning from IPv4 to IPv6. In particular, we believe that it will be a valuable tool to developers porting applications from IPv4 to IPv6. For instance, a server application ported to IPv6 can be tested without having to port the client as well.

For more information about the IPv6/IPv4 translator, its performance, and source availability, please visit our web page at:
www.cs.washington.edu/research/networking/napt

References

[1] S. Deering and R. Hinden. Internet Protocol, Version 6. RFC 1883, December 1995.

[2] P. Srisuresh and K. Egevang. The IP Network Address Translator (NAT). RFC 1631, May 1994.

[3] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 1933, April 1996.

[4] E. Nordmark. Stateless IP/ICMP Translator (SIIT). Work In Progress.

[5] J. Bound. Assignment of IPv4 Global Addresses to IPv6 Hosts (AIIH). Work In Progress.

[6] R. E. Gilligan, S. Thomson, J. Bound, and W. R. Stevens. Basic Socket Interface Extensions for IPv6. Work In Progress.

[7] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF Internet Draft, March 1998. Work In Progress.

[8] J. Mogul and S. Deering. Path MTU Discovery, RFC 1191, November 1990.

[9] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6, RFC 1981, Aug. 1996.

[10] J. Postel. Internet Control Message Protocol. RFC 792, Sep. 1981.

[11] J. Postel. Internet Protocol. RFC 791, Sept. 1981.

[12] B. Fink, 6Bone Overview and Links. http://www.6bone.net

[13] B. N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. E. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Dec. 1995.

[14] Y. Rekhter, B. Moskowitz, D. Karrenberg, and G. de Groot. Address Allocation for Private Internets. RFC 1597, March 1994.

[15] H. Custer. Inside Windows NT. Microsoft Press. 1993.

[16] R. P. Draves, A. Mankin, and B. D. Zill. Implementing IPv6 for Windows NT. Proceedings of the 2nd USENIX NT Symposium, Aug. 1998.

A. Protocol Translation Details

This appendix describes the protocol translation for both IP and ICMP headers in detail. The translation description is based on the text from [4] with minor corrections based on our implementation experience.

A.1 Translating IPv4 to IPv6 Headers

If the Don't Fragment flag is true and the IPv4 packet is not a fragment (i.e., the More Fragments flag is false and the Fragment Offset is zero) then the IPv6 header fields are set as follows:

- Version: 6
- Traffic-Class: 0 (all zero bits)
- Flow ID: 0 (all zero bits)
- Payload Length: Total Length value from IPv4 header, minus the Internet Header Length (multiplied by 4) value from the IPv4 header
- Next Header: Protocol field copied from IPv4 header. If the value of the Protocol field is 1 (ICMPv4), then substitute it with 58 (ICMPv6)
- Hop Limit: Time To Live value from IPv4 header decreased by one
- Source and Destination Addresses: Depends on address translation mechanism

If there is need to add a Fragment header (i.e., the Don't Fragment flag is false or the More Fragments flag is true or the Fragment Offset is non-zero) the IPv6 header fields are set as above with the following exceptions:

- Payload Length: Total Length minus the Internet Header Length (multiplied by 4) from the IPv4 header, plus 8 for the Fragment header
- Next Header: 44 (Fragment Header)

The Fragment header fields are set as follows:

- Next Header: Protocol field copied from IPv4 header. If the value of the Protocol field is 1 (ICMPv4), then substitute it with 58 (ICMPv6).
- Reserved: 0 (all zero bits)
- Fragment Offset: Fragment Offset copied from the IPv4 header.
- M flag: More Fragments flag copied from the IPv4 header.
- Identification: The low-order 16 bits copied from the Identification field in the IPv4 header. The high-order 16 bits set to zero.

A.2 Translating IPv6 to IPv4 Headers

With exception of the IPv6 Fragment header, all other IPv6 extension headers are ignored (i.e., there is no attempt made to translate them). For each IPv6 extension header that is ignored the Payload Length needs to be adjusted by the size of these headers before the IPv4 Total Length field is calculated.

If there is no IPv6 Fragment header the IPv4 header fields are set as follows:

- Version: 4
- Internet Header Length: 5 (no IPv4 options)
- Type of Service: 0 (all zero bits)
- Total Length: Payload length value from IPv6 header, plus the size of the IPv4 header.
- Identification: 0 (all zero bits)
- Flags: Don't Fragment flag is set to true (1), and all other flags set to false (0)
- Fragment Offset: 0 (all zero bits)
- Time To Live: Hop Limit value from IPv6 header decreased by one
- Protocol: Next Header copied from IPv6 header or last extension header; and, if the value of the Next Header field is 58 (ICMPv6), then substitute it with 1 (ICMPv4)
- Header Checksum: Computed once the IPv4 header has been created
- Source and Destination Address: Depends on address translation mechanism

If the IPv6 packet contains a Fragment header the header fields are set as above with the following exceptions:

- Total Length: Payload length value from IPv6 header, minus 8 for the Fragment header, plus the size of the IPv4 header.
- Identification: Copied from the low-order 16-bits in the Identification field in the Fragment header.
- Flags: The More Fragments flag is copied from the Fragment header and the Don't Fragments flag is set to false.
- Fragment Offset: Copied from the Fragment Offset field in the Fragment Header.

A.3 Translating ICMPv4 to ICMPv6

Echo and Echo Reply (Type 8 and Type 0): set the Type to 128 and 129, respectively.

Destination Unreachable (Type 3): for most Code values set the Type to 1, unless specified otherwise below. Translate the Code field as follows:

- Code 0, 1, 6, 7, 8, 11, and 12: set Code to 0 (no route to destination)
- Code 2: translate to an ICMPv6 Parameter Problem (Type 4, Code 1) and set the Pointer to 6, which is the IPv6 Next Header field
- Code 3: set Code to 4 (port unreachable)

- Code 4: translate to an ICMPv6 Packet Too Big message (Type 2, Code 0) and the MTU field needs to be adjusted for the difference between the IPv4 and IPv6 header sizes
- Code 5: set Code to 2 (not a neighbor)
- Code 9, 10: set Code to 1 (communication with destination administratively prohibited)
- Time Exceeded (Type 11): set the Type field to 3. The Code field is unchanged
- Parameter Problem (Type 12): set the Type field to 4 and translate the Pointer values as follows: 0-to-0, 2-to-4, 8-to-7, 9-to-6, 12-to-8, 16-to-24, and for all other ICMPv4 Pointer values set the ICMPv6 Pointer value to –1.

A.4 Translating ICMPv6 to ICMPv4

Echo Request and Echo Reply (Type 128 and 129): set the Type to 0 and 8, respectively.
Destination Unreachable (Type 1): set the Type field to 3. Translate the code field as follows:

- Code 0: Set Code to 1 (host unreachable)
- Code 1: set Code to 10 (communication with destination host administratively prohibited)
- Code 2: set Code to 5 (source route failed)
- Code 3: set Code to 1 (host unreachable)
- Code 4: set Code to 3 (port unreachable)

Packet Too Big (Type 2): translate to an ICMPv4 Destination Unreachable with code 4. The MTU field needs to be adjusted for the difference between the IPv4 and IPv6 header sizes taking into account whether or not the packet in error includes a Fragment header
Time Exceeded (Type 3): set the Type to 11. The Code field is unchanged.
Parameter Problem (Type 4): If the Code is 2 then set Type to 12, Code to 0, and Pointer to –1. If the Code is 1 translate this to an ICMPv4 protocol unreachable (Type 3, Code 2) message. If the Code is 0 then set the Type to 12, the Code to 0, and translate the Pointer values as follows: 0-to-0, 4-to-2, 7-to-8, 6-to-9, 8-to-12, 24-to-16, and for all other ICMPv6 Pointer values set the ICMPv4 Pointer value to –1.

Author Information

Marc E. Fiuczynski (mef@cs.washington.edu) is a Ph.D. student in Computer Science and Engineering at the University of Washington. His research interests are internetworking, operating systems, extensible systems, and intelligent I/O systems. He received his B.A. in Computer Science from Rutgers College in 1992 and his M.S. in Computer Science and Engineering from the University of Washington in 1995. He's worked for several years on the SPIN extensible operating system and hopes to complete his Ph.D. degree before the next millennium.
Vincent K. Lam (vkl@cs.washington.edu) is an undergraduate student in Computer Science and Engineering at the University of Washington, and graduates in June 1998 with a B.S. degree.
Brian N. Bershad (bershad@cs.washington.edu) is an Associate Professor in Computer Science and Engineering at the University of Washington. His research interests include operating systems, distributed systems, networking, parallel systems, and architecture.

Increasing Effective Link Bandwidth by Suppressing Replicated Data *

Jonathan Santos †
David Wetherall ‡
Software Devices and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology
http://www.sds.lcs.mit.edu/

Abstract

In the Internet today, transfer rates are often limited by the bandwidth of a bottleneck link rather than the computing power available at the ends of the links. To address this problem, we have utilized inexpensive commodity hardware to design a novel link layer caching and compression scheme that reduces bandwidth consumption. Our scheme is motivated by the prevalence of repeated transfers of the same information, as may occur due to HTTP, FTP, and DNS traffic. Unlike existing link compression schemes, it is able to detect and use the long-range correlation of repeated transfers. It also complements application-level systems that reduce bandwidth usage, e.g., Web caches, by providing additional protection at a lower level, as well as an alternative in situations where application-level cache deployment is not practical or economic.

We make three contributions in this paper. First, to motivate our scheme we show by packet trace analysis that there is significant replication of data at the packet level, mainly due to Web traffic. Second, we present an innovative link compression protocol well-suited to traffic with such long-range correlation. Third, we demonstrate by experimentation that the availability of inexpensive memory and general-purpose processors in PCs makes our protocol practical and useful at rates exceeding T3 (45 Mbps).

*This work was supported by DARPA, monitored by the Office of Naval Research under contract No. N66001-96-C-8522.

†Email: jrsantos@mit.edu
‡Email: djw@lcs.mit.edu

1 Introduction

In the Internet today, transfer rates are often limited by the bandwidth of a bottleneck link rather than the computing power available at the ends of the links. For example, access links (modem, ISDN, T1, T3) restrict bandwidth due to cost, while wireless links restrict bandwidth due to properties of the media. A traditional solution to this problem is the use of data compression, either at the link or application level. Existing compression schemes, however, tend to miss the redundancy of multiple instances of the same information being transferred between different clients and servers. This is problematic because such transfers have become prevalent with the growth of information services such as the Web.

Danzig's 1993 study of Internet traffic [3] noted that half of the FTP transfers could be eliminated with a caching architecture that suppressed multiple transfers of the same information across the same link. Since that time, protocols and traffic patterns have changed with the growth of the Web – it is now HTTP, not FTP, that is dominant. However, the level of redundancy is still perceived to be high, despite the application-level caching mechanisms that have emerged to curtail it.

In this paper, we revisit the problem of improving effective link bandwidth in the context of traffic with replicated data, as may occur due to TCP retransmissions, application-level multicast, DNS queries, repeated Web and FTP transfers, and so on. We have designed an innovative link compression scheme that uses a network-based cache to detect and remove redundancy at the packet level. Our

scheme takes advantage of the availability of inexpensive memory and general-purpose processors to provide an economical means of purchasing additional bandwidth. That is, given the one-time costs of $5000 per PC and the monthly costs of $2500 per T1 (1.5 Mbps), it is cheaper to purchase the two PCs used by the scheme than the bandwidth they are expected to save.

Our scheme has several interesting properties:

- It is independent of the format of packet data contents and so provides benefits even when application objects have been previously compressed, e.g., for Web images already in JPEG or GIF format.
- It utilizes a source of correlation that is not available at individual clients and servers and is not found by existing link compression schemes.
- It provides the bandwidth reduction benefits of caching in a transparent manner, e.g., there is no risk of stale information or loss of endpoint control.
- It constructs names at the link level using fingerprints and so does not depend on higher level protocol names or details. For example, the same information identified by different URLs will be compressed by our scheme, but not by Web caches.

Our scheme overlaps application-level caching systems – most notably Web caches – in that both reduce the impact of repeated transfers of the same information. However, our scheme is intended to complement Web caches rather than to compete with them, since it addresses a slightly different goal and works at a different level. For example, Web caches do not take advantage of replication across multiple caching systems, protocols and application objects.

In this paper, we present: a trace-driven traffic analysis that motivates our scheme; the design of our system; and an experimental characterization of a prototype implementation. Our traffic analysis in Section 2 uses several traces of at least one million packets each that we recorded between our site (the MIT Laboratory for Computer Science, including the Web consortium) and the rest of the Internet. In Section 3, we describe the system architecture and compression protocol, along with a prototype implementation running under Linux. In Section 4, we evaluate the performance of this prototype. We then contrast our system with related work and conclude in Sections 5 and 6, respectively.

2 Analysis of Replicated Traffic

To understand the potential of a system for suppressing replicated data transfers at the packet level, we began our design by analyzing network traffic. We define a packet to be *replicated* when the contents of its payload match exactly the contents of a previously observed payload. Since packet headers are expected to be constantly changing and a function of the source and destination hosts rather than the data being transported, we do not consider them in our search for replicated data.

Note that it is not clear that overlapping Web transfers will translate into replication that satisfies our definition and that may be detected and removed at the packet level. First, data sent multiple times must be parceled into packet payloads in the same manner, despite potentially different protocol headers, path maximum transmission units (MTUs), and protocol implementations. Second, the timescale of replication (which may be hours for Web documents) must be observable with a limited amount of storage. We therefore characterize the replication as defined above by answering the following questions:

- How much data is replicated?
- What kind of data is most likely to be replicated?
- What is the temporal distribution of replicated data?

2.1 Obtaining the Packet Traces

As input to our analysis, we collected a series of full packet traces of all traffic exchanged between our site and the rest of the Internet. New traces (rather than publicly available archives) were necessary because we require the entire packet contents in order to detect repeated data. The choice of our site was expedient, but it makes an interesting test case because it is a diverse environment hosting many

	All Inbound		Inbound HTTP		All Outbound		Outbound HTTP	
Set	Total Vol. (MB)	% Repl.	Total Vol. (MB)	% Repl.	Total Vol. (MB)	% Repl.	Total Vol. (MB)	% Repl.
A	277	12	26	19	554	18	267	24
B	189	2	13	8	563	21	384	28
C	105	2	3	10	294	21	239	24
D	237	11	22	7	606	19	420	25
E	217	4	28	8	594	23	427	29
Total	1025	7	91	11	2610	20	1736	26

Table 1: Total volume and replicated percentage (by volume) of inbound and outbound traffic

clients and servers. It includes the Web Consortium, MIT Laboratory for Computer Science and the MIT AI Laboratory.

Each trace was captured using `tcpdump` as a passive monitor listening to Ethernet traffic traveling on the segment between the Lab and the Internet. Five sets of 1-2 million packets each were gathered at different times of day, corresponding to approximately 2.6 GB of raw data in total. No packet capture loss was detected.

2.2 Analysis Procedure

We statically analyzed each trace by searching through the packets sequentially for replicated data. To expose the application data, we progressively stripped protocol headers up to the TCP/UDP level. For example, TCP payloads were identified by removing first the Ethernet, then IP and finally TCP headers. Our analysis therefore slightly underestimates the amount of replicated data due to changing headers at higher protocol layers that could not easily be taken into account; one example of traffic that falls into this category is DNS responses.

2.3 Replication by Traffic Type

Our initial analyses classified replication by traffic direction (incoming and outgoing) and type (TCP, UDP, other IP, and other Ethernet). It quickly became evident that most replication occurred in outgoing TCP data on ports 80 and 8001, i.e., Web traffic responding to queries from other sites. To highlight this, we separately classified TCP port 80 and 8001 traffic as HTTP traffic.

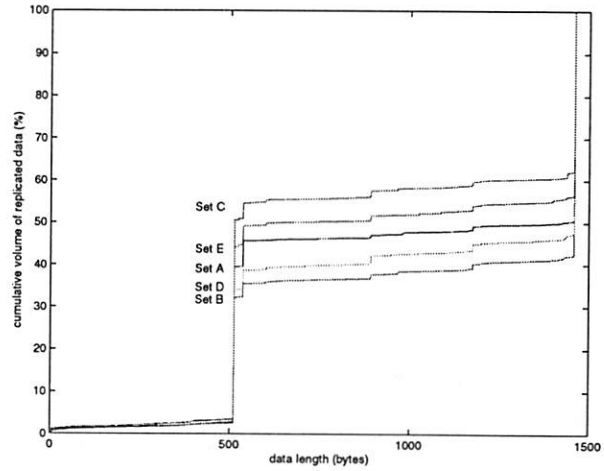


Figure 1: Cumulative volume of replicated data by packet length

Table 1 summarize the amount of replicated data that was found in each packet trace, for inbound and outbound traffic, respectively. The left-hand columns show the results for all types of traffic in each trace, while the right-hand columns summarize the replication in only the HTTP traffic for each trace.

These results support our intuition that there are significant amounts of replicated data present in the traces. Further, most of the traffic, as well as a greater percentage of replication, exists in the outbound traffic. Therefore, for the remainder of this paper, we will focus on the outbound traffic over the link.

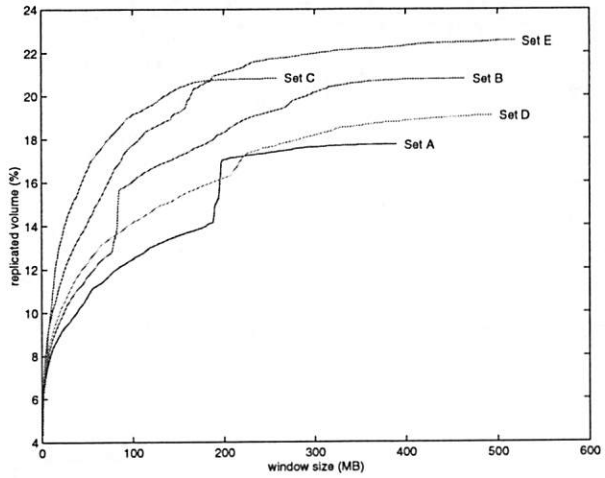


Figure 2: Percent of outbound traffic versus window size

2.4 Replication by Packet Size

A further criterion that is important to our scheme is packet size. Replication in large packets will result in a more effective system than replication in small packets when fixed-length packet overheads and packet processing costs are taken into account.

To assess this effect, we classified the replicated data according to the length of the data payload. Figure 1 depicts the cumulative volume of replicated data according to packet length. The sharp increases round 500 and 1500 bytes correspond to the default TCP segment size is 536 bytes and the maximum Ethernet payload 1.5 Kb. It is apparent that 97% of the volume of replicated data occurs in packets with a length greater than 500 bytes. This suggests that small per packet space costs required for compression will not result in a significant system overhead.

2.5 Distribution of Replication

Finally, the timescale of replication events determines the size of the packet cache needed to observe and remove such redundancy. To quantify this effect, we determined the interval, in bytes of data, from each match to the previous copy of the match. These intervals were then grouped to compute the percentage of the replicated traffic that could be identified as a function of window size.

Figure 2 shows this result for all outbound traffic. The positive result that we infer is that the majority of replicated data can be observed with a cache of 200 MB, i.e., reasonable results can be expected if we cache the data in the amount of RAM that is presently available in PCs.

3 Design and Implementation

We now describe the design and implementation of a compression architecture that suppresses replicated data based on the analysis from Section 2. The overall goal of our scheme is simply to transmit repeated data as a short dictionary token, using caches of recently seen data at both ends of the link to maintain the dictionary and encode and decode these tokens.

The correct operation of this scheme as a distributed system is complicated by the fact that messages may be lost by the channel. Our design must resolve the following issues:

- How are dictionary tokens generated?
- How are dictionaries at either end of the link maintained in a (nearly) synchronized state?
- How are (inevitable) differences in dictionary state handled?

Our approach is based on the insight that the fingerprint of a data segment is an inexpensive name for the data itself, both in terms of space and time. We are aware of the use of fingerprints for identification and version control in various systems, e.g., Java RMI/OS, but to the best of our knowledge this is the first time that fingerprints have been applied for this purpose at the network layer.

We selected the MD5 hash [12] for our implementation because it is 128 bits and may be calculated in one rapid traversal of the data; on a PentiumII (233MHz) the computational rate of fingerprinting exceeds 200 Mbps. Further, given that the hash is large enough and collisions rare enough, it is effectively a unique name for the data. For example, though our architecture handles collisions, none were detected in our trace data analysis.

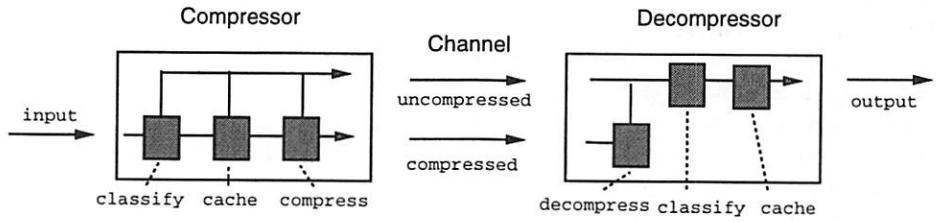


Figure 3: Components of the Architecture

To handle message loss in a lightweight fashion, we have opted to process messages independently, such that each message is the unit of error generation and recovery. That is, our scheme is connectionless (aside from the dictionary state) and does not require that a reliable transport protocol be run across the link in order to recover from errors.

3.1 Architecture

The main components of our architecture are shown in Figure 3, which shows a unidirectional compression system to simplify our description. The system consists of a compressor, a channel, and a decompressor. The compressor is a repeater (perhaps part of a router) that accepts input traffic, processes it to compress replicated data, and transmits the resulting packets over the channel. Conversely, the decompressor accepts traffic from the channel, processes it to remove compression, and transmits it as output traffic. The channel itself may be any bidirectional link; we use the reverse direction to carry protocol control messages. Bidirectional compression is achieved by using two instances of the protocol, one for each direction.

Both the compressor and decompressor are composed of several modules for classifying, caching, and compressing packets. Our architecture allows different policies to be selected for the implementation of each of these stages, subject to the constraint that compressor and decompressor implement identical processing in order to ensure that their dictionaries are closely synchronized. In particular, the dictionary caches must be of equal size. We describe each module in turn.

3.1.1 Classifying Packets

Not all packets need be entered into the dictionary cache. Our analysis in section 3 showed that most of the replicated data in our traces was composed of outgoing Web traffic and large packets. An implementation may take advantage of such bias by selectively considering certain types of traffic for cache inclusion. The classification step in our architecture serves this role, and must be performed in the same manner at the compressor and decompressor.

The classifier further encodes the rules for identifying application data units (ADUs) embedded within the payload of packets, e.g., the stripping of headers up to the TCP/UDP level. By using application level framing concepts (ALF) [2], other extension policies could be designed to cater for specific application headers or compensate for the different division of data across different protocols.

3.1.2 Caching Policies

The cache module maintains the dictionary state, naming payloads by their fingerprint. Our architecture allows any fingerprint to be used depending on the required tradeoff between speed, space and collisions. In our implementation we use MD5, though stronger fingerprints such as the SHA [10] or weaker fingerprints such as MD4 may be used.

Two policies govern the operation of the cache: the inclusion policy decides which payloads selected by classification should be admitted to the cache, and the replacement policy decides which payloads should be evicted when more space is needed. As for classification, the compressor and decompressor must implement identical policies.

Our default policies are simple: all payloads that are output by the classifier are entered into the cache, and the cache is maintained in least-recently-used order. For inclusion, an interesting policy would be to store replicated data only after its fingerprint had been encountered a certain number of times. Depending on the number of times a given payload is repeated, this may significantly reduce the storage required to suppress a given volume of replicated data. For replacement, results with Web caching [15] suggest that taking payload length into consideration may improve performance, since larger data payloads translate to higher per-packet savings.

A further issue that affects inclusion is fingerprint collision. Collisions are expected to be extremely rare, but nevertheless it is conceivable that they may occur. If so, they must not result in a deterministic error, with the same offending data being repeatedly transferred to correct perceived transmission errors.

In our architecture, collision detection is performed as part of cache lookup and insertion at the compressor. Every time a fingerprint matches, the full payload data is compared with the existing cache contents before it is entered. If a collision is encountered, the fingerprint is marked as illegal in the dictionary and the colliding payload is transmitted without any compression. Any subsequent payloads which index to the illegal fingerprint are also transmitted uncompressed. These illegal entries must persist at the compressor until the decompressor is reset.

3.1.3 Compression and Decompression

Finally, the compression and decompression modules exchange dictionary tokens to suppress the actual transfer of repeated data. Different policies may be used by the compressor to decide when to compress payloads. Our default policy is to simply send tokens whenever repeated data is available. Alternative policies may be useful when the link possesses a large latency or high error rate and it is desirable to further reduce the chance that the far end of the link does not have the payload corresponding to a token. In these cases, it would be possible to send tokens after the payload has been sent multiple times, or, in the case of TCP traffic, send the token when the acknowledgment of the payload is detected in the reverse direction.

3.2 Protocol Operation

We now describe the exchange of protocol messages between the compressor and decompressor. These fall into three cases.

- In the normal case, a payload is transferred (being entered in the dictionary as a side-effect) and after some interval another payload with the same contents is transferred, this time as a dictionary token. We refer to this case as *compression.*
- Occasionally, however, message loss on the channel may cause the two caches to lose synchronization and a dictionary token that is transferred must be returned to the sender to be resolved. We refer to this case as *rejection.*
- Further, if either the compressor or decompressor is restarted during the operation of the protocol, it is desirable to reset the other cache to a known state. Therefore, we add reset messages to the protocol.

3.2.1 Compression

The sequence of message exchange in the compression case is shown as a time sequence diagram (with time proceeding down the page) in Figure 4. These descriptions assume that the incoming packet passes the classification stage and satisfies the inclusion policy; packets that do not are simply forwarded over the link in the usual fashion.

When the compressor receives a packet {HdrA, X} to be forwarded over the link, where HdrA is the TCP/IP header and X is the data payload, it first computes H(X), the fingerprint of X. If it finds that no entry indexed by H(X) exists in its cache, it stores X in its cache, indexed by H(X). It then forwards the TCP/IP packet across the link. Upon receiving a TCP/IP packet forwarded over the channel, the decompressor also computes H(X), and stores X in its cache, indexed by H(X). The TCP/IP packet is then output from the system.

At some point later, the compressor may receive a packet HdrB, X, for which an entry indexed by H(X) already exists in its cache. This indicates that it has

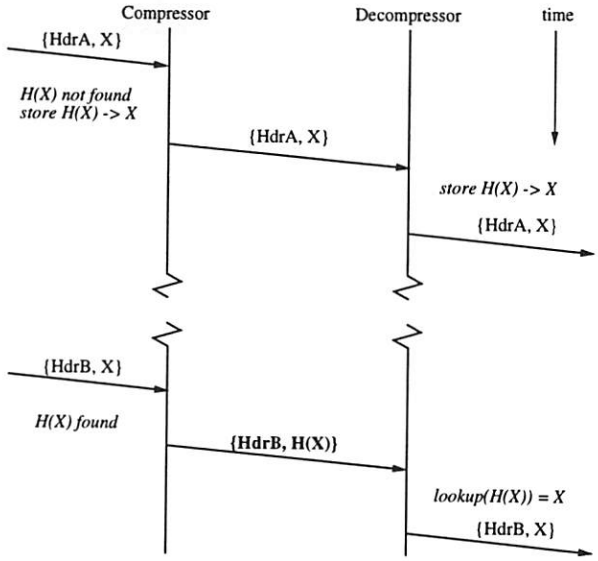


Figure 4: Compression protocol

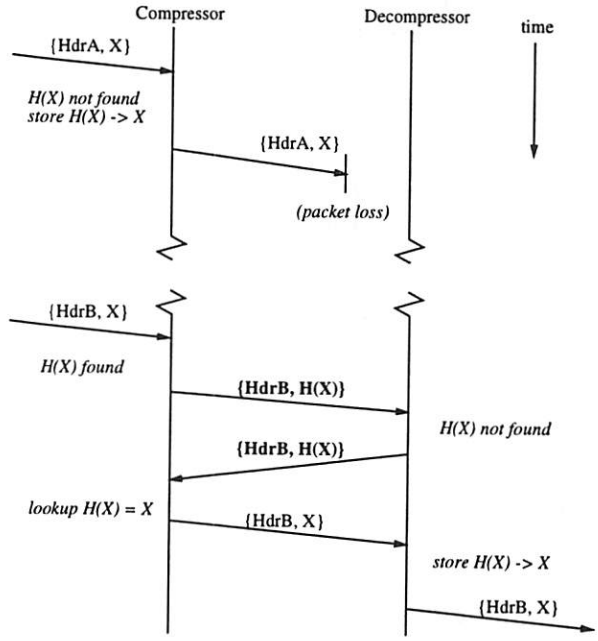


Figure 5: Rejection handling

already received a packet containing X, which it forwarded over the link. Therefore (assuming the compression policy is satisfied) it sends a packet to the decompressor containing the TCP/IP header HdrB and the fingerprint H(X). Fingerprint packets appear in bold type in the protocol diagrams.

The implementation must therefore provide a means for these "fingerprint packets" to be distinguished from ordinary IP packets. In practice, this is not a problem, because the codepoint used for demultiplexing protocols at the link level may be overloaded, e.g., we allocate additional types for the Ethernet protocol type field. Note that it is important that this identification scheme not increase the length of the packet, since this would necessitate a segmentation and reassembly protocol to accommodate maximum length datagrams.

When the decompressor receives a fingerprint packet {HdrB, H(X)}, it determines the data payload X that is indexed by H(X) in its cache. It then forwards the corresponding TCP/IP packet {HdrB, X} to the network on that end.

3.2.2 Rejection

The protocol as described above is incomplete, for it does not handle the case where a packet containing the first instance of a data payload is lost while being sent across the link. We expect this case to be rare for most channels, since bit error rates typically contribute negligibly to the overall packet loss, and loss due to congestion may be detected at the compressor (since it results in queue overflow) and the lost payloads not entered into the dictionary.

Nevertheless, if the protocol is left as is, the lack of feedback means that the compressor does not know that the decompressor never received the original payload. This means that it will send further copies of the payload by its fingerprint when the packet is retransmitted, causing ongoing loss. To correct this error, we introduce rejection handling into the protocol to handle events in which the decompressor receives a fingerprint that is not in its cache.

Figure 5 depicts rejection handling with another time sequence diagram. After message loss, if the decompressor receives a fingerprint packet {HdrB, H(X)} for which H(X) is not a valid entry in its cache, it sends the entire fingerprint packet (including the header) back to the compressor as a rejection packet. When the compressor receives this rejection, it determines the data X that is indexed by H(X). This is highly likely to be in the cache at the compressor since it was sent in the recent past. The compressor then sends the complete TCP/IP packet

{HdrB, X} to the decompressor, which processes the packet as if it were receiving a new TCP/IP packet. It therefore enters it into its cache for subsequent use.

If any of the packets that are sent as part of the rejection handling are lost, or in the unlikely event that the compressor no longer has the payload corresponding to the rejected fingerprint in its cache, then the transmission has failed, and no further steps are taken to recover. This residual loss will then be handled by the reliability mechanisms of the application in the same manner that packet loss is handled today.

3.3 Reset Messages

During normal operation of the protocol, the compressor keeps track of all illegal fingerprints (i.e., those fingerprints for which a collision occurred.) In the event that this state is lost (e.g., the compressor is restarted), the compressor reliably sends a cache reset message to the decompressor to ensure that the decompressor does not have any entries indexed by a previously illegal fingerprint.

Further, restarting the decompressor during operation of the protocol may result in significant rejection traffic. Therefore, we explicitly send a cache reset message from the decompressor to the compressor. This is merely a performance optimization, and is not essential for correctness.

3.4 Implementation

We implemented the architecture described above using a pair of Intel-based PentiumII 300MHz machines running Linux 2.0.31 with 128MB of RAM each. The machines were directly connected to each other via a dedicated 10 Mbps Ethernet and both machines were also connected to the 100 Mbps Ethernet network which comprises our research group's LAN. Both compressor and decompressor modules were written in C and ran as user-level processes.

The compressor machine was configured with IP forwarding enabled in the kernel. However, we modified the kernel forwarding routine to send the packets to the user-level program instead of handling the routing itself. We also allocated additional Ethernet protocol types to distinguish the fingerprint and rejection packets from the uncompressed packets.

We implemented the dictionary caches using hash table structures with a least-recently-used replacement strategy. For fingerprints, we used the MD5 hash of the payload. We also used a classifier that only accepted data with payloads of at least 500 bytes since Figure 1 indicates that the remaining data comprises only 3% of the replicated volume. Finally, we limited the amount of memory available for the caches, excluding the overhead induced by the hash table implementations, to 200MB each.

4 Experimental Results

To evaluate the system, we performed three sets of experiments.

- We measured the bandwidth savings that our system provides in practice when operating on real traffic.
- We measured the baseline performance of the compressor and decompressor to gauge at what rates our system may be used.
- We compared the bandwidth savings produced by our system with alternative compression schemes.

4.1 Bandwidth Reduction

Our main design goal is to reduce the amount of bandwidth consumed by replicated data. We measured bandwidth savings by inserting our system into the network at the point where we previously gathered traces; see section 2.1. We kept track of the amount of data input to and output from the system and the amount of data transmitted across the compressed channel while the system ran for 24 hours and processed approximately 50 GB.

Figure 6 shows the resulting bandwidth reduction for each minute of the run. It shows that the implementation is effective in reducing the bandwidth utilization by approximately 20% for the entire duration of the experiment.

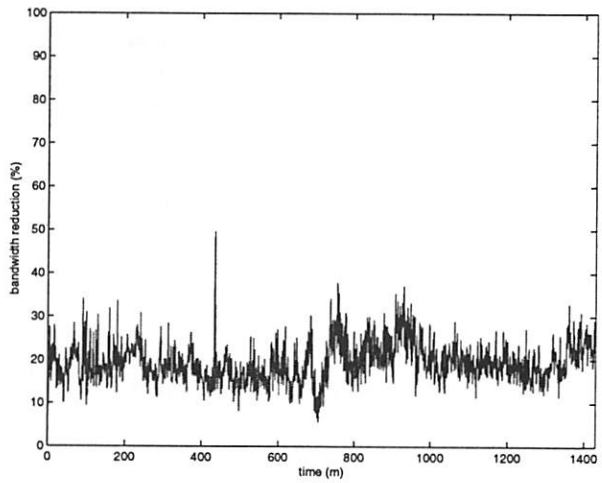


Figure 6: Bandwidth reduction for all outbound traffic

4.2 System Performance

Since we are interested in the potential of this scheme for use in higher speed networks (with capacities exceeding 10 Mbps) we measured the overall system performance to see how fast it would run.

Packet streams containing no detectable replication incur the highest amount of processing required for each packet at both the compressor and decompressor. This therefore presents the worst-case load for our system, and we used such streams to test the performance of the system.

To measure the throughput of the system, we ran the system over a 100 Mbps channel and sent our test stream of packets over a TCP connection that flowed over the channel. We measured latency by using `tcpdump` as a passive monitor to capture and timestamp packets entering and leaving both the compressor and decompressor. To observe small latencies, we use a modified Linux kernel that records timestamps using the processor cycle counter at driver interrupt time.

The results of our tests were that our implementation was capable of forwarding over 6000 packets per second with a maximum throughput exceeding 60 Mbps. Furthermore, the latencies of the compressor and decompressor were both approximately 390μs. These results are encouraging; our system can already run at rates exceeding T3 (45 Mbps), despite the fact that it is a user-level prototype that has not been tuned, e.g., to minimize copies. Further, preliminary comparisons with other compression schemes (such as gzip as discussed below) suggest that our scheme is significantly less computationally expensive. The similar and low latencies of compressor and decompressor result in a balanced system for given hardware and a small impact on overall latency. They are also likely to improve significantly with an kernel-space implementation since the overhead of context switching would be removed.

Set	Reduction	Compression	Both
A	12.08	20.09	31.30
B	15.50	24.08	37.35
C	18.81	18.42	33.90
D	14.37	17.95	32.44
E	17.90	18.58	35.32
Avg	15.73	19.92	34.07

Table 2: Percentage of bandwidth saved by Reduction and Compression (gzip) on outbound traffic

4.3 Other Compression Methods

Since bandwidth savings are heavily data dependent, we compared our bandwidth reductions with those of other compression schemes to place them in context and help gauge their significance.

As an approximation to real systems, we ran our trace data through a process that applied gzip compression to packet payloads and recorded volume statistics. To simulate useful schemes under real-time and high throughput conditions, we used the fastest library setting and processed packets individually; even so, gzip is substantially slower than our scheme and could not keep pace with a 10 Mbps link. Table 2 compares this compression with our scheme for removing replicated data. We infer from these results that our scheme provides similar benefits, somewhat smaller on average, but requiring less computation.

To look at the effects of combining our reduction and regular compression, we ran our trace data through a process that combined the two, first removing replicated data and compressing the remainder. Table 2 also shows these results. It highlights the

fact that reduction and compression combine rather than overlap, as each tends to tap a correlation on different timescales.

We also considered the impact of header compression, but quickly realized that it would provide smaller savings. With the average packet size of our trace close to 500 bytes, elimination of TCP/IP headers from all packets would save no more than 8% of the bandwidth, and this best case is unlikely to be obtained across a link where there is significant traffic mixing.

5 Related Work

We discuss two categories of related work: compression and caching.

5.1 Compression Techniques

When faced with a limited transfer rate, higher effective throughput can be obtained by compressing the data before transmitting it across a link. Some link protocols such as PPP make provisions for the negotiation of such a compression scheme between the ends of the link [13, 11]. Packets are then compressed (either individually or as a stream) when they enter the link and uncompressed at the other end. Alternatively, higher compression ratios can typically be obtained by compressing the data before sending it into the network. This is so for two reasons. First, lossless compression utilities such as gzip [5] work better with larger and unmixed inputs because of their statistical properties. Second, application-specific lossy schemes, such as JPEG [8] for photographic images, may be used. A further form of compression that is appropriate for some applications is delta-encoding, where a set of differences is transmitted instead of the complete object; Mogul et al. have shown that this technique may result in significant savings for Web traffic [9].

However, none of these types of compression remove the redundancy of transfers of the same information between different clients and servers whose paths cross within the network. Compression of application data can reduce the amount of information needing to be transferred, but by definition cannot remove redundancy across different clients and servers. Compression of data at the link level can in theory remove such redundancy, but in practice does not. This is because algorithms that build dynamic dictionaries typically limit their search to a small window of the data stream compared to the scale on which we will show that there is replicated traffic, e.g., gzip may search approximately 32KB, while we have detected significant correlation at 1000 times that scale.

Another type of compression that is frequently employed is packet header compression. Schemes specialized for compressing TCP/IP headers [7, 4] may reduce their impact by an order of magnitude in the best cases, and hence may have a significant impact on bandwidth usage when there are many short packets. In the packet traces we observed, however, the volume of headers was small compared with the volume of payloads, i.e., even eliminating all TCP/IP headers would not make as large a difference we demonstrated by suppressing data. Furthermore, compared to payload compression, header compression taps an orthogonal source of correlation, and could therefore be used in addition to other techniques.

5.2 Application-Level Caching

An alternative to compressing at the link level is for each application to construct its own system for caching its data-types. This is clearly not viable for all applications, but may be worthwhile in terms of bandwidth for popular cases such as the Web. To examine the tradeoffs, we briefly compare our scheme with Web caching using the generic configuration of Figure 7. Here, an organization is connected to the rest of the Internet by a single access link that is the bottleneck for transfers between the two domains. We ignore more costly options that reconfigure the system to shift this bottleneck, e.g., purchasing more bandwidth, spreading load over multiple links, or co-locating Web servers with the ISP.

Today's Web caches [14, 1] are deployed by organizations to reduce both client latency and wide area bandwidth requirements. A caching system may therefore be readily deployed at point A to protect incoming bandwidth by combining client requests. Existing caching systems, however, are more limited in their ability to protect outgoing bandwidth by combining server responses. Our traces show

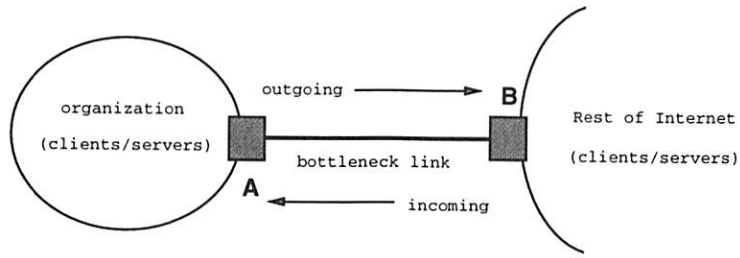


Figure 7: Generic Web Caching Configuration with a Bottleneck Link

that existing client caching has not eliminated redundant server transfers (and it is highly unlikely that this will soon be the case as it requires that all Web clients be configured in a single caching system well-matched to the underlying topology). Further, it may not be possible to place a Web cache at point B and configure the rest of the Internet to use it, since point B is typically under the control of a different organization and proxy caches require the cooperation of their clients. That is, placement of application caches inside the network may require a large degree of sharing and cooperation between users compared to the link-level solutions we have studied, which may be deployed by the network operator when and as needed to buttress weak links.

Since link layer schemes are transparent to applications, they present a different set of tradeoffs than does application-level caching. The latter may utilize application semantics, and so should be more effective for the particular application. It may also improve performance in other respects, in the same manner that Web caches lower latency as well as reduce bandwidth. However, application-level schemes may also have side-effects that a transparent scheme does not. For example, unlike Web caches, our scheme will never return stale data, nor complicate or bias server operations such as request logging. Further, because they function across all applications, link layer solutions are capable of removing redundancy across multiple protocols, e.g., FTP as well as HTTP. More interestingly, our link layer solution suppresses identical content irrespective of application names and protocol details. For example, the same Web page contents will be suppressed, even if it is named by different URLs, generated dynamically, or marked as uncacheable. This effect may be significant: Douglis et al. found such duplication to occur for as many as 18% of full-body Web responses in some traces [6].

Given these tradeoffs, we believe that our scheme complements rather than competes with application-level caching systems. Web traffic is so predominant that special-purpose caching mechanisms must become ubiquitous in order to distribute load and build a scalable Internet.

Our scheme provides protection at a lower level and across changing application and traffic patterns. It can thus be applied to portions of the network selectively, e.g., to bottleneck access and long-haul backbone links, and will remove the replication that remains after application-level caching.

6 Conclusions and Further Work

In this paper, we have presented a innovative link compression protocol that suppresses the transfer of replicated payloads. We have demonstrated that, despite existing caching mechanisms, there is a significant amount of replicated traffic that is amenable to detection and reduction by our scheme. The protocol itself works by maintaining (nearly) synchronized packet caches at either end of a link and sending repeated payloads that are encountered as fingerprints. We have further shown by experimentation that the protocol is lightweight enough to be implemented on commodity hardware at rates exceeding T3 (45 Mbps). For real packet traces the increase in available bandwidth from our scheme can be around 20%. This makes it an economically viable option for increasing available Internet access bandwidth.

In addition to the bandwidth savings we realized, our scheme is significant in several respects:

- Unlike other compression methods, it is independent of the format of packet data contents, and so provides benefits even when application objects have been previously compressed.
- It utilizes a source of correlation that is neither available at individual clients and servers nor found by existing link compression schemes, and hence can be used in addition to other link compression schemes.
- It provides the bandwidth reduction benefits of caching in a transparent manner, e.g., unlike Web caching there is no risk of stale information or loss of endpoint control.
- Unlike Web caching, it does not depend on particular protocols, client configuration or application names; it may thus be useful as a general-purpose mechanism to protect links from redundant transfers (which have many sources) as applications and traffic patterns change.

Finally, we see several areas that would benefit from further work:

- Implementation techniques such as different cache insertion and replacement policies that improve the range of match detection for a given amount of storage would improve the value of the system.
- The impact of the protocol on performance should be characterized across a range of bit error rates to confirm that it does not exacerbate packet loss.
- Additional classification techniques that increase the amount of data that we are able to detect as replicated, which would improve the effectiveness of the system as a whole.

Acknowledgments

We thank our fellow members of the Software Devices and Systems group. In particular, we wish to acknowledge Vanu Bose for insightful discussions and feedback throughout the course of this research, and John Guttag for his support and guidance. We also thank Mark Handley and Max Poletto for proofreading.

References

[1] A. Chankuntod et al. A Hierarchical Internet Object Cache. In *USENIX'96*, 1996.

[2] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM '90*, 1990.

[3] P. Danzig et al. A Case for Caching File Objects Inside Internetworks. In *SIGCOMM '93*, 1993.

[4] M. Degermark et al. Low-loss TCP/IP Header Compression for Wireless Networks. In *MOBICOM'96*, 1996.

[5] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. Request For Comments: 1951, May 1996.

[6] F. Douglis et al. Rate of Change and other Metrics: a Live Study of the World Wide Web. In *USENIX Symp. on Internetworking Technologies and Systems*, 1997.

[7] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. Request For Comments: 1144, February 1990.

[8] I. JTC1/SC2/W10. *Digital Compression and Coding of Continuous-Tone Still Images.* IEC Draft International Standard 10918-1, 1992.

[9] J. Mogul et al. Potential benefits of delta-encoding and data compression for HTTP. In *SIGCOMM '97*, 1997.

[10] NIST. Secure Hash Standard. FIPS PUB 180-1, May 1993.

[11] D. Rand. The PPP Compression Control Protocol. Request For Comments: 1962, June 1996.

[12] R. Rivest. The MD5 Message-Digest Algorithm. Request For Comments: 1321, April 1992.

[13] W. Simpson (Ed.). The Point-to-Point Protocol. Request For Comments: 1661, August 1994.

[14] D. Wessels. The Squid Internet Object Cache. http://squid.nlanr.net/Squid/, 1997.

[15] S. Williams et al. Removal Policies in Network Caches for World-Wide Web Documents. In *SIGCOMM '96*, 1996.

Making Commodity PCs Fit for Signal Processing

Michael Ismert
Software Devices and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
izzy@lcs.mit.edu, http://www.sds.lcs.mit.edu/

Abstract

Commodity PCs are on the verge of being capable of performing a variety of signal processing tasks that previously required special purpose hardware. Advances in the speed and width of their processors and internal buses allow these machines to manipulate data at rates that would allow their users to interact with a diverse range of sampled media, such as the raw RF spectrum and ultrasound. However, today's PCs lack an I/O system capable of delivering the appropriate bandwidth to these signal processing applications. These applications demand high *continuous* throughput I/O that smooths the inter-sample jitter introduced by interrupts, I/O bus latency, scheduling latency, etc. This paper presents a system that provides this functionality.

Our system is composed of two tightly integrated parts: a PCI device that provides high raw I/O bus throughput and operating system enhancements to manage the device and provide low overhead transfers across the boundary between kernel and user space. The performance is excellent, providing up to 512 Mbits/sec of continuous throughput for an application. A description of both parts of the system is given, along with performance measurements,and a brief description of an application.

1 Introduction

Processors in commodity PCs have reached the point where they are capable of performing the signal processing tasks typically left to special-purpose hardware[SPL92]. The possibility of moving these tasks from external hardware into software running on the main CPU presents some exciting possibilities[TB96]. These include multi-purpose devices where new functions are possible by adding or upgrading software, allowing PCs to emulate devices ranging from cellular or cordless phones to ultrasound or EKG monitors[Tha97, BCT97]. This also allows rapid deployment of new or upgraded standards as well as the ability to rapidly prototype new signal processing algorithms or protocols. Finally, this approach can enable increased performance in several ways. The obvious one is to take advantage of the fact that market factors are driving rapid improvements in PC performance. However, this approach also allows the integration of the signal processing with higher-level applications, presenting the opportunity for system-wide optimization, as well as allowing the signal processing functions to be dynamically modified based on measurements of changing system characteristics in order to improve performance.

There are already industry movements in this direction. Vendors of software modems, for example, advocate digitizing the phone line signal and doing the necessary coding and modulation in high-priority interrupt handlers[Tra97]. Today's PCs can handle these low data rate tasks easily. However, for the more aggressive applications we envision, such as those encompassing large bands of the RF spectrum, the off-the-shelf PC is not yet a viable platform.

Increasing clock rates, superscalar architectures, and SIMD techniques such as Intel's MMX allow general-purpose processors to dedicate a reasonable number of cycles to each data sample. Along with the processors, the internal buses have increased in both speed and width, allowing high-bandwidth data to be moved about within the PC. However, today's commodity PCs are lacking two important

pieces that enable this sort of signal processing. The first is an I/O device capable of digitizing (and perhaps downconverting) a generic wide-band signal and delivering the samples to the PC's main memory (and performing the reverse operation as well). The second is the ability of current commercially available operating systems to deliver high-bandwidth, low-jitter, continuous data streams to the application.

This paper presents a system that fills in these gaps in current commodity PCs. There are several goals that our system must satisfy. First, it must provide high *continuous* throughput between the digitizing front-end and the applications. As an example, the A-side cellular telephony band is 12.5 MHz wide. When digitized at 25 MHz with a sample size of 16 bits, the data rate necessary to transfer this stream of samples to the application for processing is 400 Mbits/sec.

Second, the system must provide the applications with what appears to be a jitterless sample stream. In standard signal processing systems based on dedicated digital hardware or DSPs, the incoming samples arrive at a constant rate and are processed with a fixed delay between when a sample enters the system and when the output based on that sample leaves the system. The processing happens in lock step with the I/O, so the DSP is guaranteed that it will have a constant stream of regularly spaced samples on which to do processing. In a personal computer, however, there are no such simple guarantees. Virtual memory, multiple levels of caching, and competition for the I/O and memory buses add jitter to the expected amount of time required for a sample to travel from an I/O device to the processor. In addition, using a multi-tasking operating system ensures that the signal processing application will not always be the active process, which adds jitter to the rate at which samples are processed. The jitter introduced by all of these sources must be smoothed out.

Since many signal processing applications of interest, such as those involving two-way voice communications, are sensitive to latency, the system must not introduce an excessive amount of delay between the time when samples enter the system and when they are processed. In addition, signal processing applications are also processing-intensive. As a result, the system must accomplish the previous goals with as little processing overhead as possible.

Finally, we imposed the goal of simplicity on the system, both on its use and design. Application programmers should not have to use a new set of system calls to allocate memory for buffers, transfer data, and access the device, nor should the system design require such things to be implemented.

The system is composed of two co-designed parts: a hardware component described in section 2, and operating system support described in section 3. The performance of the system is presented in section 4. Section 5 is a discussion of the previous work related to this area.

2 The I/O Device

The General Purpose PCI Interface (GuPPI) provides the necessary high-speed I/O port for PC-based signal processing. It is a two board sandwich, consisting of a full-size PCI card which contains the engine for moving samples to and from memory, and a A/D card which is capable of sampling an analog input at 40 million samples/second with a 12 bit resolution and generating an analog output from a sample stream of the same rate and resolution.

A block diagram of the data transfer portion of the GuPPI is shown in figure 1. The core of the GuPPI is the Xilinx FPGA, which contains the PCI controller and DMA engine. The design provides high raw bandwidth between the A/D card and the PC's main memory.

The A/D card is directly connected to a set of FIFO buffers in both the input and output directions. These FIFOs provide the buffering necessary to absorb jitter caused by the bursty access to the PCI bus. In the input direction, the FIFO holds incoming samples until the GuPPI acquires the bus and can transfer them into memory. In the output direction, the FIFO holds excess samples that have been transfered from memory but are waiting to be accepted by the daughter card.

As well as absorbing jitter, the FIFOs also serve to temporally decouple the timing of the GuPPI, which operates using the PCI clock, from the daughter card, which typically operates using the A/D sampling clock. This eliminates the need for designers of daughter cards to worry about synchronization or metastability issues. It also separates the data

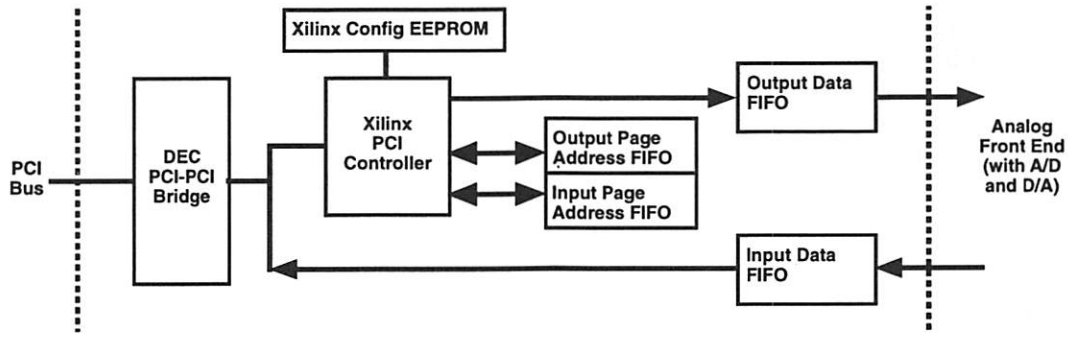


Figure 1: GuPPI Block Diagram.

transfer and processing from the fixed rate realm of the analog front-end, allowing it to be bursty.

The GuPPI is connected to the PCI bus for its high throughput. PCI is an example of how the internal buses in commodity PCs have improved enough to allow the movement of high speed data; its raw capacity is just over 1 Gbit/sec, which puts it well over the bandwidth needed for our initial target applications. Several other possible points of connection were possible, such as the ISA bus or the I/O bus used in some other architecture, such as the SBus. However, the low bandwidth of the ISA bus (approximately 140 Mbits/sec) eliminated it. The SBus, while providing reasonable bandwidth, limited the GuPPI to operation in Sun workstations. The PCI bus is a more popular I/O bus architecture that is common to all commercially available PCs, including Intel and AXP-based machines, and provided easy potential upgrades to double speed, double width, and mobility via CardBus.

The GuPPI is a PCI bus master; that is, it has the ability to initiate transfers on the PCI bus. As such, the GuPPI can DMA sample streams to/from main memory at high speed with minimal intervention from the processor, which supports our goal of low overhead. The GuPPI implements a new variant of scatter/gather DMA that we have named *page-streaming*. The GuPPI has two page address FIFOs, one each for input and output, that hold the physical page addresses associated with buffers in virtual memory. At the end of a page transfer, the GuPPI reads the next page address from the head of the appropriate page address FIFO and begins transferring data to/from it. Using the programmable FIFO flags, the GuPPI triggers an interrupt when the supply of page addresses runs low, and the page addresses are replenished by the interrupt handler in the device driver.

There are several benefits for signal processing applications from using page streaming DMA. First, since the processor replenishes the addresses, the GuPPI only uses its bus grants to transfer data, which results in more efficient use of the PCI bus by the GuPPI. Since DMA transfers are always page-length and page-aligned, no buffer length information needs to be transferred across the bus, reducing the bus overhead associated with the DMA. In addition, page streaming simplified the design of the GuPPI by reducing the complexity of Xilinx DMA controller. Page streaming also has several nice characteristics related to the operating system enhancements, which are described in the next section. The lone drawback to page streaming is that transfer sizes are fixed to page-sized, page-aligned buffers; however, the continuous nature of the sample streams ensures that transfers of this size can always be completed, and page alignment can be easily arranged by manipulating the beginning of buffers.

3 Operating System Enhancements

The operating system support consists of a device driver for the GuPPI and several small additions to the virtual memory system, all for the Linux kernel[1]. The total size of the code is just under 1200 lines, with the virtual memory system additions representing just 200 of those. Another important aspect of the additions is that they do not affect the performance or functionality of any part of the system not related to the GuPPI; all other applications

[1]The Linux kernel version used is 2.0.31.

run completely unperturbed. The device driver provides for the continuous transfer of data between the GuPPI and main memory while absorbing jitter due to the scheduling of the signal processing applications. The virtual memory additions provide low overhead, high-bandwidth transfer of data between the application and the device driver.

3.1 GuPPI Device Driver

The device driver is responsible for using the raw burst performance of the GuPPI to provide a continuous stream of I/O to the application, and to absorb jitter caused by interrupts and the scheduling of other processes.

Initially, a purely user-level approach, similar to [vEBBV95], was implemented and tested. The GuPPI's control and status registers were memory-mapped into the virtual memory space, allowing the application to completely control the GuPPI. Application-allocated buffers were locked down in physical memory and given to the GuPPI for DMA. While the burst performance was excellent, the continuous performance suffered, particularly during periods where either another process received a substantial portion of the processor time or an interrupt for another device occurred and a slow interrupt handler was called. These performance breaks were due to the inability of the application to keep the DMA engine supplied with buffers to be transferred. Ironically, the process was being swapped out while it was attempting to provide the GuPPI with the buffers necessary to keep the I/O continuous in the event that it was swapped out.

The failure of the user-level approach to be able to maintain a continuous I/O stream of samples motivated the use of an interrupt-driven device driver within the kernel. This driver made the virtual memory system additions (described later) necessary. The implementation and performance of the kernel-level driver are presented in this paper.

3.1.1 Input

The input portion of the GuPPI device driver uses a ring of buffers into which samples are transfered. These buffers are part of the kernel virtual memory space, but are locked down in physical memory. The driver initially fills the input page address FIFO with addresses from buffers at the head of the ring. When this FIFO drains to a certain (programmable) level, an interrupt is triggered, and the interrupt handler queues up more buffers from the head of the ring. The level that triggers the interrupt is set such that there are sufficient pages remaining to absorb samples arriving before the interrupt handler can provide new buffers.

When an application reads data from the driver, it is given the buffer at the tail of the ring. Rather than copy the data from the kernel to the user buffer, the driver uses the virtual memory additions to swap the buffer provided by the application for the buffer in the kernel[2]. The swapping not only avoids the cost of copying the data, but eliminates the need for the kernel to allocate a buffer to replace the one given to the application.

The ring buffers allow the driver to absorb scheduling jitter for those applications attempting to run in real-time. When the signal processing application is not scheduled, the head of the ring, which is the last buffer given to the GuPPI for DMA, will move away from the tail of the ring, which is the next buffer to be given to the application. When the application is once again scheduled, it will need to be able to read and process buffers faster than the GuPPI can fill them, moving the head back towards the tail. If the application cannot do this, it has no chance of maintaining any real-time operation. The amount of scheduling jitter that the system can absorb is dependent on the size of the ring.

3.1.2 Output

The output portion of the GuPPI driver maintains a queue of buffers that the application has written to the driver. The total size of this queue is bounded to keep applications from using all of physical memory. Initially, the driver fills the GuPPI's output page address FIFO with addresses from buffers at the head of the queue. Similar to input, when this FIFO drains to a certain level, an interrupt is triggered and the interrupt handler replenishes the FIFO from buffers at the head of the queue, if they exist.

Our early experience with writing applications has

[2]The application can configure the size and number of buffers used in the ring; this is the means by which the application knows the size of the buffer to provide to the `read` system call.

shown that it is usually efficient to generate output waveforms into buffers on the fly. This allows the output portion of the driver to use the same virtual memory additions to swap application buffers with kernel buffers of the same size. In order to avoid constantly allocating new buffers to swap back to the application, the driver maintains a queue of recently-used buffers and only allocates new ones when a buffer of the proper size is not in the cache. Since applications tend to use buffers of the same size, this works quite well.

The output queue plays the same role for output as the ring buffers do for input. While scheduled, the application should be able to put several buffers on the waiting queue. If the application is running faster than the GuPPI, these buffers will not be depleted by the next time the application is again scheduled to run. There is a maximum total memory size that the driver allows to be on the waiting and sending queues; writes will return unsuccessfully when this point is reached.

3.2 Virtual Memory Additions

The standard Unix approach, when faced with a new device, is to implement a Unix device driver to control the device and transfer data to and from it. However, the performance problems associated with using the default Unix I/O system to move data across the kernel/user boundary are well known. To avoid the expense of performing the data copy, previous research efforts have relied on different schemes using virtual memory manipulation and/or shared memory[DP93, CP94, And95, vEBBV95, BS96, Pai97].

Our approach to this problem uses virtual memory manipulation but, unlike most of the related work, avoids adding new buffering and I/O semantics. Systems such as those presented in [DP93, CP94, Pai97] require the use of a new I/O API in order to access the high-performance I/O system, including a new set of system calls, data structures, etc. In order to maintain a simple and familiar interface for the user, the system was designed to use the standard Unix API and copy semantics. However, virtual memory manipulations are used to make the `read` and `write` system calls to the GuPPI copy-free.

To this end, facilities have been added to the virtual memory system to swap a buffer in an application's virtual memory space with a buffer in the kernel's virtual memory space. The choice to swap buffers rather than share them between the kernel and the application had several motivating factors. For input (reading), the application may want to perform in-place computation on the samples in that buffer, so the kernel cannot turn around and reuse the buffer as part of the ring until it sure that the application has no further need for it. This involves either an addition to the API for the application to inform the kernel that a buffer can be reclaimed, some amount of data copying to keep the buffer from being overwritten, or allocating a new buffer to become part of the ring. These options all involved a significant increase in design complexity without a clear improvement in performance. For output (writing), since the applications generate data on the fly, there is no real advantage to sharing between the application and the kernel.

The ability to easily perform this buffer swap is due to the unframed, continuous nature of the data streams; enough data can be acquired such that all buffers that the GuPPI uses are page aligned and an integral number of pages long. This makes the virtual memory swap almost trivial as long as the user provides a buffer of the proper size. Each page in the user buffer is faulted into a distinct physical page. The physical addresses in the page table entries for the kernel and user buffers are then swapped, and the appropriate TLB entries in the memory management system flushed. The kernel and user buffers maintain their same protections, and there is no violation of protection because only kernel buffers that are guaranteed to contain only new incoming data (in the case of input) or stale output data (in the case of output) are given to the user. The performance gain of this swap over a kernel-to-user data copy is two orders of magnitude; more detail will be presented in section 4.

The use of page streaming DMA coupled with the virtual memory swapping has some nice characteristics. First, the constraint of page-aligned, integral page-length buffers for both operations means that buffers used for DMA are ideally suited for transfer to the application, and vice versa. In addition, since the physical addresses for both the user and kernel buffers are determined during the swap, the list of pages used to be used for the page streaming DMA is generated for free.

4 Performance

The performance numbers presented in this section demonstrate the effectiveness of our system. The experiments were all performed with warm caches on an unloaded 200 MHz Pentium Pro system running Debian Linux. The CPU on-chip cycle counter was used to measure throughput and the processing requirements generated by the GuPPI. Due to our lack of a front-end for transmit, most of the applications which have been developed are receive only, and so the receive performance has been studied significantly more.

4.1 Throughput and Overhead

The maximum rate at which an application using the GuPPI driver can maintain a continuous flow of input samples from the GuPPI is 512 Mbits/sec. This number was determined using an application that only accessed enough samples per input buffer to verify data continuity. This number provides an upper bound on the possible throughput that an application can achieve using the GuPPI. At rates above this point there is insufficient depth in the input data FIFO on the GuPPI to absorb jitter due to the PCI bus, but a new revision of the GuPPI will increase the maximum continuous throughput. For output samples, the maximum throughput is only 260 Mbits/sec. The output data FIFO is half the size of the input data FIFO, and so is not able to absorb as much PCI bus jitter; increasing this FIFO size will similarly increase the output throughput.

Input	Output
930	850

Table 1: Raw GuPPI Performance (Mbits/sec)

In order to provide high continuous throughput, the GuPPI must have higher raw throughput. Table 1 shows the maximum input and output burst performance of the GuPPI. These numbers reflect the amount of time required for the GuPPI to DMA approximately 1.2 MB of data. The measurements include only the amount of time required to DMA the data, not the time required to write page addresses into the appropriate page address FIFO. The maximum PCI throughput available is 1056 Mbits/sec, so the GuPPI is coming reasonably close to saturating the workstation's PCI bus. The lower maximum throughput for output is due to the larger latency required to initiate a read from main memory to the GuPPI.

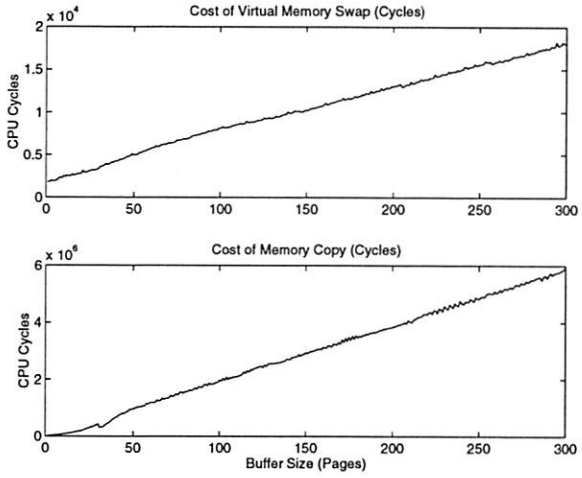


Figure 2: Virtual Memory Swap vs. Memory Copy

The performance benefits of using a virtual memory swap rather than a data copy to move samples from kernel to user buffers are shown in figure 2. The values for both cases are the average number of cycles over 100 buffer reads from the GuPPI. The same user buffer is recycled for each read, so after the first iteration all the pages in that buffer have been faulted into physical memory. Since it is expected that applications will frequently reuse their input buffers, this should not be a performance issue.

Figure 3 shows the average processing overhead imposed on the workstation by using the GuPPI to generate input. This measurement reflects the number of cycles required to perform the `read` system call (which includes the virtual memory swap) and the number of cycles required to handle interrupts generated by the input page address FIFO running near empty. The average number of cycles per sample is very low; for standard buffer sizes that applications might handle (30 to 300 pages), the processing overhead of the GuPPI is less than a half cycle per sample.

4.2 Latency

Another performance issue to consider is that of latency. By adding buffering, both FIFOs and I/O kernel buffers, to smooth the jitter in the system, we have added to the delay between when samples

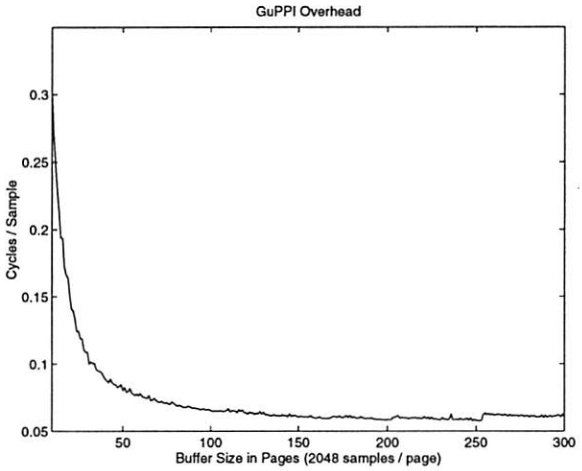


Figure 3: GuPPI Processing Overhead (Input)

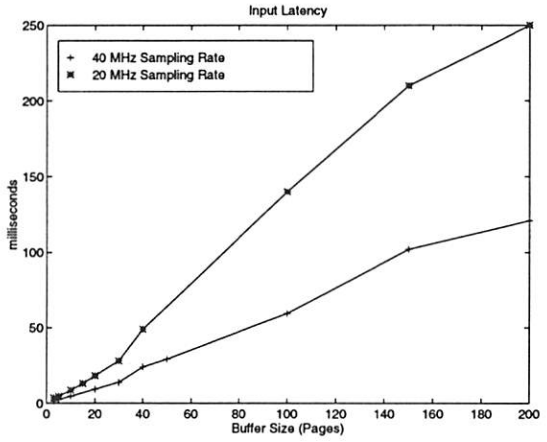


Figure 4: GuPPI Latency (Input)

are generated by the A/D converter and when they are processed. Figure 4 shows a graph of the input latency as a function of buffer size. To generate these values, a sampled step function was sampled by the GuPPI and streamed into memory. An application monitored the incoming buffers for the step and strobed a pin on the parallel port when it was detected.

The input latency of the system is heavily dependent on the sampling rate; larger buffer require more time to fill with samples before they can be delivered to the user, while higher sampling rates cause buffers to fill more quickly. The graph shows the linear relationship between the buffer size (with ring size fixed at 4 buffers) and the latency, as well as the effect of reducing the sampling rate in half; the numbers are consistent with our expectations.

The minimum input latency measured was just over 1 millisecond for a two page buffer. While the latency measured for smaller buffer sizes is acceptable for a two-way voice application or a point-to-point data connection, it is not sufficiently small for multiple-access schemes such as CSMA or TDM. In these cases, some additional hardware may be necessary to synchronize the transmitter with the receiver.

4.3 A Real Application

Along with these abstract measurements, we were also able to measure the performance of an actual real-time signal processing application that used our I/O system. This application is an AMPS cellular telephony receiver using a data rate of 204.8 Mbits/sec (8 bit samples at 25.6 MHz). The application selects a single AMPS channel from a 10 MHz wide band, demodulates it, and plays the audio to the workstation speaker in real-time. The application is processor-limited; our I/O system is able to supply it with the sample stream for the entire cellular band but the application cannot currently demodulate more than one channel. On the original 200 MHz Pentium Pro platform, the application was not even able to meet the full AMPS-specified channel selection filter because the amount of processing required caused it to quickly fall behind the data provided by our I/O system. However, the entire system has been ported to a 533 MHz AXP-based system which can meet the required specifications for one channel. So, there is still a bottleneck in our system, but it has shifted from I/O to processing, allowing us to ride the curve of improved processor performance. It is important to note that, although specialized DSPs can ride this same performance curve, they do not track it nearly as well. DSP clock rates are just reaching 200 MHz; Digital has had AXP processors which operate at 600 MHz for quite some time now. In addition, porting the I/O system and the AMPS application to the AXP system required less than a week, something which would have taken significantly longer moving from one DSP architecture or generation to another.

5 Related Work

5.1 Data Acquisition Devices

For many years, data acquisition cards for PCs have been available; these have been capable of storing samples on-board in a circular buffer until some trigger condition is met, and then dumping the contents of the buffer into memory on the workstation. These cards have typically been ISA bus devices, with no support for the direct streaming of samples into the PC's memory. Recently, however, several companies have released PCI variants of these data acquisition cards which are capable of streaming samples directly into memory[Ins98, Sci98]. These devices are capable of high burst data transfers; the Gage CompuScope 8500 is capable of bursts of 933 Mbits/sec. However, these devices are still based on the trigger/capture model; after a trigger, the device transfers samples into PC memory rather than onboard memory for some fixed period of time. In addition, these devices are input-only devices, and do not have the operating system support required for continuous high throughput to/from the application.

5.2 Host Adapter Research

Most of the research into high performance host adapters is related to network adapters. [DDP94] presents the experience of integrating Osiris, one of the first experimental ATM host adapters, with the high performance I/O work presented in [DP93]. The authors describe the ability to tune the programmable logic on the host adapter in order to simplify the task of writing efficient operating system software. Our approach to designing the GuPPI and its operating system support is very similar; however, we had the luxury of designing both the hardware and the software concurrently, and so very little tuning was actually necessary. There are also several similarities between the operation of the Osiris board and the GuPPI. Both boards implement fixed-size DMA transfers, both boards use a FIFO queue, fed by the host processor, to provide buffers for input or output, and both boards use an interrupt-driven mechanism for replenishing input buffers. However, the need to support network data rather than sample streams causes different functionality in the two boards. The performance of this pairing of host adapter and I/O system is quite high, with the ability to transfer up to 516 Mbits/sec from the adapter to the host and 325 Mbits/sec in the opposite direction.

[DPC97] presents another high-performance ATM adapter. The paper describes a technique for performing zero-copy data transfers between the host adapter and user buffers. The authors are developing a special ATM protocol that provides page aligned, integral page length data in each packet. This is essentially the same approach as that used by the GuPPI, except the continuous input stream removes the need for any special protocol.

5.3 Operating System I/O Research

A significant amount of research has been directed at optimizing the data transfer across the kernel/user boundary. [BS96] provides a taxonomy for the various ways in which the kernel/user boundary can be crossed, as well as discussing possible optimizations, implementing them in the Genie system, and providing some quantitative analysis of the various methods. The operating system modifications for the GuPPI provide facilities that are similar to the emulated copy semantics described in this paper, with some optimization based on the application characteristics.

[DP93] investigates the use of virtual memory remapping applied to system-allocated buffers, called *fbufs*, as a means of efficiently moving data across protection domains. Several optimizations are applied to this strategy, including caching and shared memory. IO-Lite[Pai97] is a system that attempts to unify all I/O related buffering and caching into a single efficient, copy-free system. IO-Lite is based on aggregates of the fbufs and uses identical methods (virtual memory remapping, shared memory) to move buffers made of these aggregates between domains. Either of these systems could replace the emulated copy mechanism as the mechanism for crossing the kernel/user boundary. However, the need for a new I/O API to support them makes them unattractive for our use.

Universal Continuous Media I/O (UCM I/O)[CP94] and Container Shipping[And95] are two more systems that seek to provide more generic I/O support. These systems also use virtual memory manipulations to avoid copying, but with move, rather

than share or copy semantics; output data disappears from and input data appears in the application's virtual memory space. Several optimizations are presented in each, including the recycling of virtual memory information (page tables, addresses, pages), the elimination of unnecessary zero-filling of allocated buffers, and the selective mapping of only the used portions of buffers. The move semantics make this sort of system unattractive in our context for two reasons. First, the loss of ring buffers in the driver reintroduces the overhead of allocating new buffers to replace those moved to the application. Second, the loss of the application's buffers makes the output buffer reuse that is characteristic of many of the signal processing applications impossible without a data copy.

UCM I/O also provides buffers shared between the kernel and the application as part of the support for continuous media. To use this support, the application allocates a ring of buffers and attaches them to a control buffer. The kernel uses a clock interrupt to schedule the I/O of the next buffer in the ring. The approach used to support input from the GuPPI is similar to the UCM I/O's continuous media support, but with some important differences. First, the ability to use the GuPPI's interrupts rather than a fixed period clock interrupt allows the system to adapt to variance in the time required to fill a buffer. Swapping the kernel ring buffers with the buffer provided by the application allows the application to hold the contents of a buffer without the contents being bashed by the kernel when the ring wraps around. In the output direction, the need to reuse the output buffers in arbitrary orders by the application makes using a ring buffer system virtually impossible without adding data copies.

6 Conclusion

The GuPPI and its operating system support close the gaps preventing today's PCs from being an effective platforms for intensive signal processing. By designing the hardware and software in tandem, we were able to create a system which is small and low in complexity, yet achieves excellent performance for the applications for which it was intended. Together, the hardware and software provide an application-level interface that simplifies the construction of software radios and related applications by making the analog front-end appear to be a conventional Unix device.

The GuPPI hardware provides a high bandwidth, low latency connection between an analog front-end and the I/O bus of a PC; it makes use of a new variant of scatter/gather DMA, page-streaming that simplifies the hardware and provides for efficient use of the PCI bus. The software drives the GuPPI, smooths jitter, and makes the data transferred by the hardware accessible to applications in user space. While essential, the operating system support needed to make this functionality available to applications is minimal, and has been implemented in such a way as to have no impact on the functionality provided to other applications.

For future commodity systems, we imagine that hardware similar to the GuPPI, coupled with an appropriate, flexible front-end, would be an invaluable I/O device to integrate into a system, with the possibility of becoming as ubiquitous as serial and parallel ports are today. In addition, we have demonstrated that the necessary operating system support to make this functionality available to applications is minimal, and can be implemented in such a way as to have no impact on the functionality provided to other applications. Software modems, while a step in the right direction, are only focussed on the limited bandwidth of a telephone line and will have little application beyond it; a system like the GuPPI would continue to enable interesting and useful applications for many years to come.

Acknowledgements

The author would like to thank the many individuals who have influenced and contributed to the development of this project. First, David Tennenhouse and John Guttag for their guidance; Vanu Bose for providing a testbed for this work; and Matt Welborn, Alok Shah, and Andrew Chiu for finding bugs by using the system for their work. This research was supported by the Advanced Research Projects Agency under contract No. DABT-6395-C-0060 (monitored by US Army, Fort Huachuca) and by equipment grants from Intel Corporation.

References

[And95] Eric Anderson. *Container Shipping: A Uniform Interface for Fast, Efficient, High-bandwidth I/O.* PhD thesis, University of California, San Diego, 1995.

[BCT97] Vanu G. Bose, Andrew G. Chiu, and David L. Tennenhouse. Virtual Sample Processing: Extending the Reach of Multimedia. *Multimedia Tools and Applications*, 5(3):259–276, November 1997.

[BS96] Jose C. Brustoloni and Peter Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 277–291, Seattle, WA, October 1996. USENIX.

[CP94] Charles D. Cranor and Gurudatta M. Parulkar. Universal Continuous Media I/O: Design and Implementation. Technical Report TR 94-34, Washington University Department of Computer Science, December 1994.

[DDP94] Bruce S. Davie, Peter Druschel, and Larry L. Peterson. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of SIGCOMM '94*, September 1994.

[DP93] Peter Druschel and Larry Peterson. Fbufs: A High Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, Asheville, NC, December 1993.

[DPC97] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of IEEE Infocom 1997*, April 1997.

[Ins98] National Instruments. NI5102 PCI Digital Oscilloscope Data Sheet, 3 98.

[Pai97] Vivek S. Pai. IO-Lite: A Copy-free UNIX I/O System. Master's thesis, Rice University, January 1997.

[Sci98] Gage Applied Sciences. CompuScope 8500/PCI Hardware Installation and Reference Manual, March 1998.

[SPL92] Lawrence C. Stewart, Andrew C. Payne, and Thomas M. Levergood. Are DSP Chips Obsolete ? Technical Report CRL 92/10, Cambridge Research Labratory, Cambridge, MA, 1992.

[TB96] David L. Tennenhouse and Vanu G. Bose. The SpectrumWare approach to Wireless Signal Processing. *Wireless Networks*, 2:1–12, 1996.

[Tha97] Samir R. Thadani. A Software-Based Ultrasound System for Medical Diagnosis. Master's thesis, MIT, May 1997.

[Tra97] Mike Tramontano. Host Signal Processing: A New Way to Communicate. Technical report, Motorola ISG, 1997.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 3-6 1995.

The Eclipse Operating System: Providing Quality of Service via Reservation Domains

John Bruno, Eran Gabber, Banu Özden and Abraham Silberschatz
Bell Laboratories, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974
{jbruno, eran, ozden, avi}@research.bell-labs.com

Abstract

In this paper, we introduce a new operating system abstraction called *reservation domains*, and describe its implementation in Eclipse, an experimental operating system that provides a testbed for Quality of Service (QoS) support for applications.

Reservation domains enable explicit control over the provisioning of system resources among applications in order to achieve desired levels of predictable performance. In general, each reservation domain is assigned a certain fraction of each resource (e.g., 25% CPU, 50% disk I/O, etc.). Eclipse implements reservation-domain scheduling of multiple resources. It currently supports CPU and disk and physical memory (working set size) scheduling.

Eclipse implements a new scheduling algorithm, Move-to-Rear List Scheduling (MTR-LS), that provides a *cumulative service guarantee*, in addition to fairness and delay bounds. Cumulative service guarantee is necessary for ensuring predictable aggregate throughput for applications that require multiple resources. Preliminary experiments indicate that MTR-LS provides good QoS in overloaded systems. In particular, MTR-LS favors *less-greedy* processes.

The Eclipse operating system is based the Plan9 from Bell Labs, and can run any Plan9 application without modification. Eclipse emphasizes the use of per-process name space, and it can schedule any I/O device or user level file system without any change to device driver or file system code.

1 Introduction

New multimedia applications, which require support for real-time processing, are pacing the demand for operating system support for Quality of Service (QoS) guarantees. The desire to support multiple real-time applications on a single platform requires that the operating system have the ability to provision system resources among applications in a manner that achieves the desired levels of predictable performance. Moreover, computer networks are starting to provide QoS guarantees with respect to packet delay and connection bandwidth. These QoS guarantees are of little use if they cannot be extended to applications executing at the endpoints. Current general-purpose multiprogrammed operating systems do not provide QoS guarantees since the performance of a single application is, in part, determined by the overall load on system. As a result, many users prefer to use stand-alone systems with limited dependency on shared servers in order to achieve some semblance of QoS by indirectly controlling the system workload. Real-time operating systems are capable of delivering performance guarantees such as delay bounds, but require that applications be modified to take advantage of the real-time features.

Our goal is to provide QoS guarantees in the context of a general-purpose multiprogrammed operating system, without modification to the applications, by giving the user the option to provision system resources among applications in order to achieve the desired performance levels.

This paper introduces a new operating system abstraction called *reservation domains*, which is intended to provide predictable QoS by controlling resource provisioning. The basic idea behind reservation domains is to isolate the performance of reservation domains from each other. In particular, time-sensitive processes can coexist with batch processes on the same system.

We discuss the importance of *cumulative service* guarantee, which complements other QoS parameters such as *delay* and *fairness*. Informally, guaranteed cumulative service means that the scheduling delays encountered by a process on various resources do not accumulate over the lifetime of the process. In other words, a process that is competing for resources will execute at a predictable rate, which is determined by the fraction of the resources that is reserved for that process, regardless of the intensity of the competition for the resource.

We continue with the description of a new schedul-

ing policy called Move-To-Rear List Scheduling (MTR-LS), that provides cumulative service guarantees, as well as fairness and delay bounds.

Next we describe the Eclipse operating system, which is a testbed for operating system support for QoS. Eclipse implements reservation domains and MTR-LS scheduling. Eclipse can schedule multiple resources independently. The current implementation can schedule CPU, disk I/O and physical memory (working set). Experimental results indicate that MTR-LS outperforms the standard Plan9 priority scheduler and a weighted round-robin scheduler. We also include an excerpt from the Eclipse implementation of MTR-LS, which illustrates the simplicity of the implementation.

The remainder of this paper is organized as follows. In Section 2 we discuss related work. Section 3 introduces the concept of a reservation domain. Section 4 presents the QoS parameters of interest. Section 5 presents the MTR-LS scheduling policy and its properties. The implementation of the Eclipse operating system is described in Section 6. Section 7 presents the results of our preliminary experiments with Eclipse. Finally, Section 8 describes our future work and conclusions. The Appendix contains a code excerpt from the Eclipse implementation of MTR-LS.

2 Related Work

Stride scheduling [16] and lottery scheduling [17] attempt to provide each process with a share of the server in proportion to its corresponding weight (number of "tickets"). Start-time fair queuing [4] and earliest eligible virtual deadline first [13] guarantee fairness bound. Class based queuing (CBQ) [15] uses bandwidth reservations to schedule packets on shared data links. Processor capacity reserves [7] provides each process with its reserved share and delay guarantees. The SMART scheduler [8] provides predictable performance for real-time applications, which may executed concurrently with non real-time applications. SMART allows processes to specify their scheduling constraints, and provides dynamic feedback to applications when the constrains cannot be met. The Rialto operating system [5] provides resource reservations similar to Eclipse. However, both SMART and Rialto schedule only CPU. The Nemesis operating system [6] provides accurate accounting of resource consumption by running the application and most of the kernel code in the same address space. Nemesis is similar to Eclipse by providing predictable performance via allocation of CPU and disk I/O to domains. However, Nemesis employs a radically different OS structure, which necessitates rewriting of most applications and device drivers. The above algorithms and systems were not designed with our cumulative service measure in mind, so it is not surprising that the properties they do enjoy are not sufficient to guarantee cumulative service effectively.

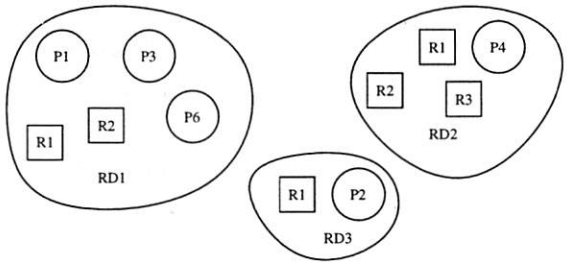


Figure 1: A Reservation-Domains System

3 Reservation Domains

In order to incorporate QoS into operating systems, we introduce the notion of a *reservation domain*. A reservation domain is a collection of processes and corresponding resource reservations. A computer system may run several reservation domains and provide several types of resources (e.g., CPU, disk, network, physical memory), which are reserved and scheduled independently. The processes that belong to a particular reservation domain are guaranteed to receive at least their reserved portions of the domain's associated resources. Figure 1 illustrates a system that runs three reservations domains: RD_1, RD_2 and RD_3. Each domain contains an explicit resource reservation for the resources $R1$, $R2$ and $R3$. One or more processes may run within a single reservation domain.

Reservation domains are designed to combine the advantages of real-time, time-sharing, and stand-alone systems. Benefits of reservation domains, which are impossible to achieve with priority based scheduling or real-time scheduling algorithms, are:

- Provides QoS guarantees even when the system is overloaded. In fact, a reservation domain is similar to a smaller, dedicated machine. Application programs need not be rewritten to use real-time services in order to deliver predictable QoS in a shared environment.
- Allows division of resources according to a policy. For example, two reservation domains may each reserve half the CPU, although one of them contains more processes than the other, and all processes are CPU bound.
- Supports interesting ways of controlling the computing environment. For example, the windowing system may be able to change resource reservations of domains associated with the window in focus (important) and of closed windows (less important). Another possibility is a resource super-

visor, that adapts resource reservations dynamically according to the "complaints" of processes. A process that misses its deadlines will complain, which will prompt the supervisor to increase its reservations. A process that meets its deadlines will keep silent, and will get a smaller reservation.

A variety of applications can benefit from reservation domains. For example, soft-real-time applications can use bounded response times, hosting applications can benefit from provisioning the system resources, and long running OLAP (on-line analytical processing) applications can use protection between reservation domains to complete their tasks in a timely manner without impacting current workload. Reservation domains can also benefit PC users, who would like to run several applications concurrently, when the applications are written with the assumption that they run on a dedicated machine.

4 QoS Guarantees

The reader who is familiar with QoS parameters and with [2] may skip this section. In this section we introduce three Quality of Service (QoS) parameters: cumulative service, delay bound and fairness. Cumulative service is a new QoS parameter, while delay bound and fairness are well known, and many scheduling algorithms do provide them. We start with an example that shows the need for the cumulative service guarantee. The definition of fairness and delay bounds will be given at the end of this section.

Example 1: We assume that the system runs one I/O bound process together with n infinite loops. The system employs a weighted round-robin scheduler, similar to the one described in Section 6.2. The I/O bound process issues I/O requests sequentially. It requires 1 msec. of CPU in order to issue the next I/O request. Each I/O request requires 23 msec. to complete. There is no contention on the I/O device. We reserve 0.5 of the CPU to the I/O bound process. We expect that each iteration of the I/O bound process will last $2 + 23$ msec., where 2 msec. is the expected execution time of the CPU phase of each iteration (1 msec. on an idle machine is equivalent to 2 msec. elapsed time when the process is reserved 0.5 of the CPU). Thus the expected execution rate of this program is $1000/25 = 40$ iterations per second.

However, whenever the I/O bound process arrives at the CPU, it is appended to the end of the ready processes queue, so it has to wait for at least n time slices before it can run. Thus the actual execution time of each iteration is $1 + n\delta + 23$, where δ is the time slice length. For example, for $n = 10$ and $\delta = 10$ msec. the execution rate of the I/O bound process is $1000/(1+100+23) \approx 8$, which is one fifth of the expected rate!

Note that the round-robin scheduler provides fairness and delay bounds, since all processes are scheduled within $n + 1$ time slices. However, the execution rate of the I/O bound process can be arbitrarily low, since round-robin does not provide a cumulative service guarantee. □

We will need the following definitions for the rest of the section: A system is considered to be a collection of resources (servers). Each resource is modeled by a "service rate" and a "preemption interval" Δt. Δt is the minimum time that the resource must run before a preemption can occur. A resource with a zero preemption interval can preempt a process at any time.

A process is considered to execute an ordered set of of *phases,* where a phase is a resource-duration pair, (s, t), where s is one of the system resources and t is the amount of time it would take resource s to complete the phase running alone on the resource. The phases of a process are not known in advance. The identity of the next phase is known only after executing the previous phase.

Even though we are interested in the performance of our system over all resources, it turns out that, due to the definition of the cumulative service guarantee, it is sufficient to study the performance at a single resource. From the point of view of resource s, a process is denoted by an ordered set of phases that alternate between resource s and *elsewhere*. The "elsewhere" resource represents the phases of processes at resources other than s.

The motivation for reservation domains is to isolate the each reservation domain from the others. We would like to guarantee a lower bound on the performance of a reservation domain, which is independent of other reservation domains. Let α_i be the fraction of a resource allocated to a reservation domain D_i. Ideally, each reservation domain D_i should receive at least α_i fraction of the resource whenever D_i has a process requiring the resource (a.k.a., whenever D_i is *busy*). We refer to the minimum service time received in the *idealized* service model as *virtual service time.* We denote the virtual service time received by reservation domain D_i in any real time interval $[\tau, t]$ by $v_i(\tau, t)$. Similarly, we denote the *real service time* (running on the resource) received by reservation domain D_i in any real time interval $[\tau, t]$ by $s_i(\tau, t)$. Note that $\alpha_i v_i(\tau, t) = s_i(\tau, t)$.

Realizable scheduling policies require that we run at most one process at a time on the resource. This means that if there is more than one process waiting to run on the resource, then one or both of the processes will experience (queuing) delay. We define $w_i(\tau, t)$ be the cumulative *real waiting time* (blocked by other domains' processes running on the resource) obtained by domain D_i in the interval $[\tau, t]$. By definition, $w_i(\tau, t) + s_i(\tau, t)$ is the total *real time* spent by domain D_i at the resource

in the interval $[\tau, t]$ either by running or waiting.

In the idealized model, receiving $v_i(\tau, t)$ virtual service time takes at most $v_i(\tau, t)$ real time. With realizable scheduling policies, in order to provide a performance as good as the one in the idealized model, the total real time $w_i(\tau, t) + s_i(\tau, t)$ spent by domain D_i at the resource in the interval $[\tau, t]$ to receive $v_i(\tau, t)$ virtual service time should be less than or equal to $v_i(\tau, t)$. We express the real system's ability to match the performance of the idealized system in terms of a cumulative service guarantee [2].

Definition 1: We say that a scheduling policy provides a *cumulative service guarantee* if there exists a constant K such that for all domains D_j and $\tau \leq t$, we have $v_j(\tau, t) \geq w_j(\tau, t) + s_j(\tau, t) - K$. □

Although the definition of cumulative service guarantee is in terms of a single resource, it implies a "global" cumulative service guarantee (using cumulative virtual service time and cumulative real time over all resources) in the multi-resource case where there is a constant number of resources [2]. Guaranteeing cumulative service is vital for applications that require multiple resources and arrival of a phase on a resource depends on the departure of previous phases. Cumulative service guarantee is necessary to ensure a predictable aggregate throughput over all the resources for such applications.

We will define the delay guarantee for the case when at any given time there is only one process within a reservation domain. For the more general case, when there can be multiple concurrent processes within a domain, the delay a phase experiences also depends how the reservation domain schedules its phases. This case requires elaborate treatment, and will not be covered in this paper.

Definition 2: A scheduling policy provides *delay bound* if, for a phase of any domain D_j, the real waiting time plus service time to complete the phase takes at most a constant amount more than d/α_j, where d is the duration of the phase. □

The fairness parameter measures the ability of the system to ensure that domains that are simultaneously contending for the same resource will "share" that resource in *proportion* to their reservations, independent of their previous usage of the resource [3]. That is, a fair scheduling policy does not penalize a domain that utilized an idle resource beyond its reservation when other domains become busy on that resource.

The definition of fairness is based on another idealized service model called *processor sharing* [11]. Under processor sharing, each domain receives a service proportional to its fraction on a resource. Since ideal processor sharing cannot be implemented in practice, fairness is defined as:

Definition 3: A scheduling policy is *fair* if there exists a constant F such that for any time interval $[\tau, t]$ during which a pair of domains, D_i and D_j, both continuously require the resource, we have $|s_i(\tau, t)/\alpha_i - s_j(\tau, t)/\alpha_j| \leq F$. □

Reservation domains specify their QoS requirements by providing a service fraction α_i for each system resource. Admission control ensures that the sum of the service fractions of all domains do not exceed certain prescribed limits. Admission control is necessary for delay and cumulative service guarantees. It is not required for fairness.

5 Move-to-Rear List Scheduling

This section presents the Move-To-Rear List Scheduling (MTR-LS) policy, which provides a cumulative service guarantee, is fair, and has bounded delay. MTR-LS policy is a generalization of the one presented in [2]. The main difference is that we allow here the aggregate quantum allocated for a domain to be partitioned. This improves QoS guarantees and average delay and waiting time. In the following subsections we present the MTR-LS policy, its complexity, and its properties.

5.1 Algorithm and Data Structures

Central to the MTR-LS policy is an ordered list, $\mathcal{L}$, of pairs *(i,left)* where i is the index of a reservation domain (i.e., D_i) and *left* is size of the *quantum,* which is the maximal amount of service time reservation domain D_i can receive without being interrupted. There can be multiple occurrences of domain D_i on $\mathcal{L}$, that is, pairs appearing at different positions in list $\mathcal{L}$ which have the same first coordinate i. The pairs *(i,left)* will be called *tokens* in the following discussion.

We say that domain D_i is before D_j on $\mathcal{L}$ if the first token of domain D_i on $\mathcal{L}$ appears before all tokens of D_j on $\mathcal{L}$. Each domain, whether it is busy or idle, has at least one token on $\mathcal{L}$. The system defines the *service cycle* as the constant T, which is the upper bound on the sum of all the quanta represented in list $\mathcal{L}$. The sum of quanta in $\mathcal{L}$ that belong to domain D_i is equal to $\alpha_i T$. We require that $\alpha_i T$ must be an integer.

MTR-LS performs the procedure shown in Figure 2 at every *decision epoch.* Decision epochs occur at the time of the arrival of a new domain, the departure of a domain, the expiration of the current quantum, the completion of the phase of the current running process (e.g. the process blocks), or the end of the current preemption interval. In other words, a currently running process may be preempted only at the end of the current preemption interval (Δt) or at the end of the current quantum.

Figure 3 depicts the domain update routine, which is called every decision epoch to update the token of currently running domain. This routine will either split the current token into two, leaving one token in the cur-

```
Decision_Epoch()
    stop elapsed counter;
    if a new domain D_i is admitted to the system then
        append the token (i, α_i T) to list L;
    if a domain D_i is removed from the system then
        remove all tokens belonging to D_i from L;
    if state == busy then
        Update_Domain();
    Run_a_Domain();
```

Figure 2: Decision Epoch Processing

rent place and appending the other to the end of $\mathcal{L}$, or move the token entirely to the end of $\mathcal{L}$, depending on the remaining quantum in the token. In any case, the sum of the quanta in the two tokens is equal to the quantum in the original token. Figure 4 shows the routine Run_a_Domain, which selects the next domain to run on the resource. The counter **elapsed** records the elapsed time to the next decision epoch.

The service obtained by a process may be less than the allocated quantum due to the termination of a phase or the arrival of a process. In the former case, the phase terminates, the process goes elsewhere, and the first runnable domain on $\mathcal{L}$ is serviced next. In the latter case, if the arriving process belongs to a domain which is ahead of the current running process's domain in the list $\mathcal{L}$, then the running process is preempted (as soon as the preemption interval permits) and a process of the first runnable domain on $\mathcal{L}$ is serviced next.

Figure 5 shows the Combine_Elements routine, which combines neighboring tokens belonging to the same domain. It is called whenever the list $\mathcal{L}$ is changed. This routine has to examine $\mathcal{L}$ only in the vicinity of the last change (removal or addition of tokens).

Example 2: Figure 6 illustrates the operation of MTR-LS. There are three tokens on $\mathcal{L}$, which belong to domains D_1, D_2 and D_3. The processes P_1, P_2 and P_3 are associated with domains D_1, D_2 and D_3, respectively. The total quanta of D_1, D_2 and D_3 are 10, 5 and 15, respectively. Initially (Figure 6a) all the domains are busy. Since the first busy domain is D_1, process P_1 is executed. P_1 goes "elsewhere" after receiving seven time units of service. $\mathcal{L}$ is updated such that quantum of D_1

```
Update_Domain()
    Let (i,left) be the current token being serviced in L
    Append (i,elapsed) to the end of list L;
    left' ← left - elapsed;
    if left' == 0 then
        remove current token (i,left) from L
    else
        replace current token with (i,left');
    Combine_Elements();
```

Figure 3: Domain Update

```
Run_a_Domain()
    if there is no runnable domain on the list L then
        state ← idle;
    else
        Let (j,left) be the first runnable token in L;
        state ← busy;
        run D_j on the resource for at most left time
          units (current quantum);
        start elapsed timer;
    wait for next decision epoch;
```

Figure 4: Run a Domain

is partitioned as shown in Figure 6b. Now the first busy domain is D_2. Therefore, process P_2 is executed. Once the quantum of D_2 is exhausted in five unit of time, D_2's token is placed at the tail of the list (Figure 6c). The next busy domain is D_3. Therefore, process P_3 selected for execution. After three units of time, domain D_1 becomes busy (process P_1 arrives at the resource). Since D_1's token precedes D_3's token in the list, P_3 is preempted. The quantum of D_3 is partitioned as shown in Figure 6d, and process P_1 is scheduled. □

5.2 Complexity

In a simple implementation of MTR-LS, a domain may have multiple tokens on $\mathcal{L}$. Each token corresponds to a quantum of a given size and the sum of all the quanta for a given domain is equal to $\alpha_i T$. The list $\mathcal{L}$ should support the following operations: 1) find the first busy domain in $\mathcal{L}$; 2) split a token, and append one of the parts at the end of $\mathcal{L}$; the original token is either updated in place or deleted; 3) append tokens of a new domain; 4) delete all tokens of a domain; and 5) combine tokens in $\mathcal{L}$. All of the operations except the first may be performed in constant time. The first operation may take $O(T)$ if all domains have $\alpha_i T$ tokens of one time quantum each.

The worst case running time of MTR-LS may be improved by keeping the domains in a heap instead of in a list, marking the tokens by an increasing time-stamp, and sorting the domains by the lowest time-stamp they currently have. The complexity of this implementation is $O(\log n)$ where n is the number of busy domains. Further time reduction can be achieved by using a priority queue or other data structures that are described in [14].

5.3 Properties of the MTR-LS Policy

The MTR-LS policy provides fairness, delay bound and cumulative service guarantees. The proofs for these properties can be found in in [1]. In particular, MTR-

```
Combine_Elements()
    if (i, m_1) and (i, m_2) are consecutive in L then
        replace them with (i, m_1 + m_2)
```

Figure 5: Combines tokens in $\mathcal{L}$

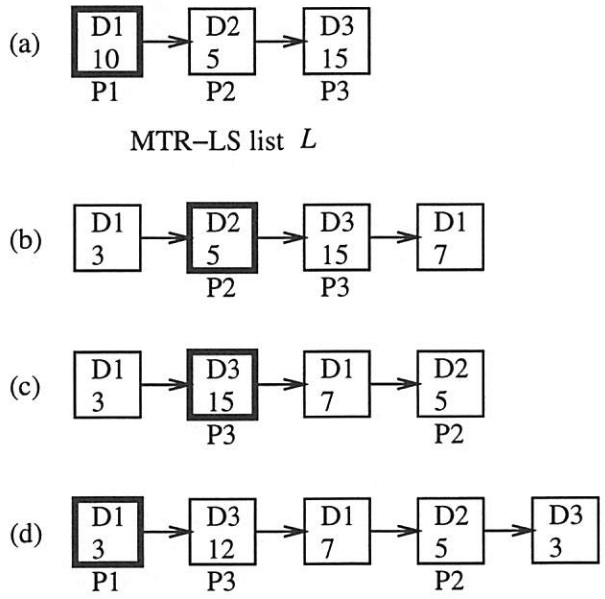


Figure 6: Example of MTR-LS Scheduling

LS policy is fair with a bound of T. MTR-LS provides cumulative service guarantee with a bound of T if $\sum_i \alpha_i \leq 1$ and the preemption interval is zero, which means that a running process is preempted immediately by a ready process that is ahead of it in $\mathcal{L}$. However, if the preemption interval is positive, MTR-LS requires that $\alpha_j \leq 1 - \sum_i \alpha_i$ for all domains D_j in order to provide a cumulative service guarantee with bound T. The reason is that a ready process may be delayed by the currently running process until the end of the current preemption interval. The delay bound provided by MTR-LS follows directly from the cumulative service guarantee (Section 4).

6 The Eclipse Operating System

Eclipse is derived from the Plan9 operating system from Bell Labs [12]. It provides a per-process name space, access to resources via the name space, a file access protocol (9P), that provides a uniform interface to all servers, and many interesting file servers, such as the windowing system $8\frac{1}{2}$, an FTP file system, etc. [10].

Eclipse provides reservation domain functionality on top of Plan9, without any change to existing applications or servers. Eclipse is compatible with Plan9; that is, Eclipse retains the external interface of Plan9 (system calls, protocols, name space structure, etc.).

Eclipse implements reservation and scheduling of CPU, I/O and physical memory (working set size), which we describe in Sections 6.2, 6.3, and 6.4, respectively. The resources are managed independently. Resource reservations and reservation domain management can be done by the shell; no systems level programming is needed.

name	access	description
`rdcpu`	r/w	% CPU reservation of current RD
`rdcpusum`	r	sum of current CPU reservations
`rdcpulim`	r	maximum allowed CPU reservation
`rdio`	r/w	% I/O reservation of current RD
`rdiosum`	r	sum of current I/O reservations
`rdiolim`	r	maximum allowed I/O reservation
`rdmem`	r/w	physical memory reservation in KB
`rdmemsum`	r	sum of physical memory reservation
`rdmemlim`	r	available physical memory
`rdsched`	r/w	current CPU scheduling algorithm

Table 1: Control Files for Resource Reservation

Since the sum of the explicit reservations of a resource $\sum_i \alpha_i$ may be less than one, Eclipse distributes the unreserved portion of the resource evenly among all reservation domains. For example, the effective reservation of domain i is $\alpha_i + (1 - \sum \alpha_i)/n$, where n is the number of reservation domains.

6.1 Managing Reservation Domains

Eclipse maintains a list of all active reservation domains (RDs). Each process belongs to a single reservation domain. A new process inherits the reservation domain of its parent, unless the process is created by `rfork(RFPROC|RFRDG)`, which places the process in a newly created reservation domain. A process may remove itself from its current reservation domain and start a new reservation domain by an appropriate call to `rfork`. In fact, each process may belong to a different reservation domain. A new reservation domain is created without any explicit reservations. However, the new domain is assigned an effective reservation, that includes its proportional share of the unreserved portion of the resources, as described above.

Resource reservation is accomplished by writing strings to the appropriate control files. Table 1 describes some of the control files. For example, the shell command `echo 50 > /dev/rdcpu` requests reservation of 50% of the CPU to the reservation domain which runs this command. Eclipse will deny resource reservations if total reservation exceeds the corresponding limit. Eclipse deletes a reservation domain and releases its resources as soon as the last process belonging to this reservation domain is terminated.

6.2 CPU Scheduling

Eclipse implements MTR-LS and weighted round-robin scheduling (WRR), which can be selected by the `rdsched` control file. For example,

`echo w > /dev/rdsched` selects WRR, and `echo m > /dev/rdsched` selects MTR-LS.

Weighted Round-Robin (WRR) is similar to round-robin scheduling, but the CPU slice of each process is proportional to its reservation. In our case, the CPU slice of the processes belonging to a particular reservation domain is proportional to the domain's reservation. Note that WRR scheduling without an explicit CPU reservation is identical to regular round-robin.

Eclipse implements WRR scheduling by maintaining a a *cpu_time_left* field for each reservation domain, which contains its remaining CPU reservation for the current *CPU service cycle*. Eclipse maintains two ready queues: *run_hi* and *run_lo*. Ready processes are appended to *run_hi* if the corresponding reservation domain has a *cpu_time_left* > 0. Otherwise, the process is appended to *run_lo*. Eclipse selects the first process from *run_hi* that has an associated *cpu_time_left* > 0. Requests with *cpu_time_left* ≤ 0 are moved to *run_lo*. The clock interrupt handler decreases the corresponding *cpu_time_left*, appends the current process to either *run_hi* or *run_lo* according to the above criterion, and calls the scheduler. The *cpu_time_left* of all reservation domains is renewed at the beginning of every CPU service cycle according to the corresponding CPU reservation. At this time all processes are moved from the *run_lo* to the *run_hi* queue.

Eclipse does not maintain a global tokens list $\mathcal{L}$ for MTR-LS, rather it keeps the creation time-stamp for each token, and sorts the reservation domains by the lowest time-stamp of the tokens that they hold. In this way, the list $\mathcal{L}$ is replaced with a collection of cyclic buffers, one for each reservation domain, which keep the tokens of that domain. Each reservation domain consumes its tokens in their time-stamp order. The Appendix contains code excerpts from the Eclipse kernel that illustrate the implementation of MTR-LS. This implementation scans the ready process list, and selects the ready process that is associated with the reservation domain that holds the token with with the lowest time stamp. The advantage of this implementation is its simplicity. However, the time complexity of each scheduling decision is $O(n)$, where n is the number of ready processes. A more sophisticated implementation may scan the ready domains queue instead, which may reduce the scheduling time. The overhead of maintaining the tokens during a clock interrupt is constant.

The current Eclipse implementation uses clock interrupts rate of 200Hz. The CPU service cycle (T) is 1/2 second long. More frequent clock interrupts improve responsiveness at the expense of more frequent context switches.

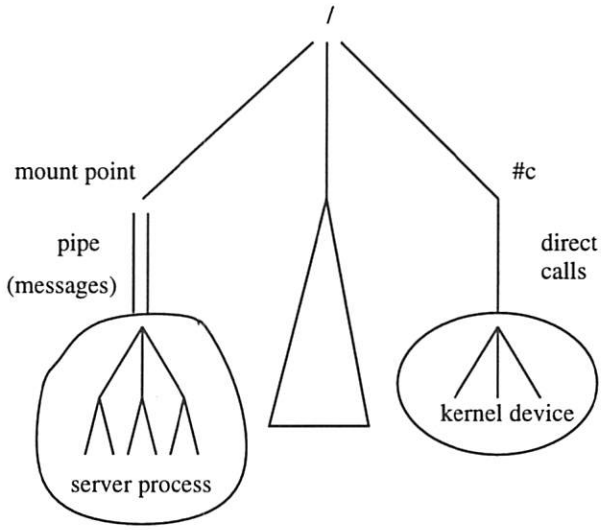


Figure 7: Eclipse/Plan9 Name Space

6.3 I/O Scheduling

Eclipse assumes that I/O devices handle a series of non-overlapping requests (one request at a time), although servers may queue several requests internally. This assumption is adequate for disks with single arms, but grossly inaccurate for network devices, which can handle several full duplex streams of interleaved packets.

The implementation of I/O scheduling is closely related to the structure of Eclipse/Plan9 name space, as illustrated in Figure 7. The name space is formed by joining independent hierarchies of files, which are provided by file servers and kernel devices. File servers are user level processes, which respond to 9P requests over bi-directional pipes. Kernel drivers are directly called by the kernel, which avoids the 9P protocol overhead.

Only a part of the name space may be subject to I/O scheduling, according to a flag in the `mount` and `bind` system calls. All other parts of the name space are not affected by I/O scheduling. This is essential, since all resources in Eclipse are part of the name space. For example, it is inappropriate to schedule requests to screen and mouse. Moreover, an attempt to schedule paging requests may cause a deadlock.

Eclipse implements I/O scheduling by intercepting `read` and `write` system calls, as illustrated in Figure 8. The rectangles inside each reservation domain denote resource reservations, and the circles denote processes. Since all I/O requests pass through the same system calls interface, I/O scheduling can be applied to all clients and all servers, without any change to either. Moreover, clients and servers may not be aware that they are subject to I/O scheduling. Scheduling can be applied to both file servers and kernel devices with equal ease, without any change to them.

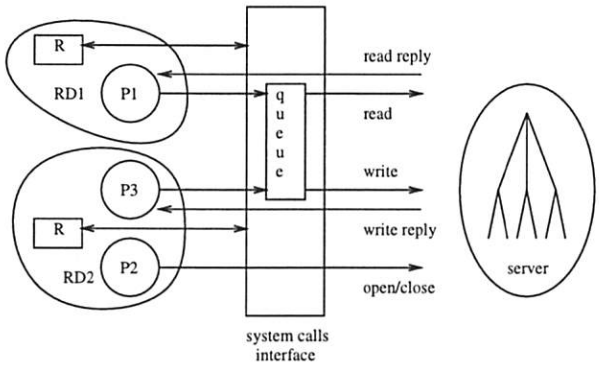


Figure 8: Implementation of I/O Scheduling

I/O scheduling is implemented similar to WRR CPU scheduling, by maintaining two I/O queues, *io_hi* and *io_lo*, and an *io_time_left* field for each reservation domain, which is reduced by the elapsed time of the request. Since the server may re-order the requests, the elapsed time is computed from the end of the previous request (if the previous request ended after the current request was delivered to the server). The reservations are renewed at the beginning of every I/O service cycle, and the requests are moved to the *io_hi* queue.

Eclipse allows the server to queue up to *maxactive* requests internally, since performance is generally improved when servers have some internal queuing (e.g., by sorting disk accesses to reduce arm movement). The current implementation has a single global requests queue. Future versions will have a queue for each server. An I/O service cycle is 1/2 second long and *maxactive* is 4.

The above scheduling algorithm is bypassed if the server is currently idle or if there is no contention (all active requests belong to the same reservation domain).

6.4 Memory Scheduling

Eclipse implements reservation of physical memory by a selective page-out strategy. The pager process scans the entire memory, and selects pages which where not accessed since the last scan and belong to reservation domains that exceeded their reservation of physical memory. This strategy ensures that the working set size does not fall below the reservation, and all pages above the reservation are subject to global LRU replacement.

6.5 Source Code

Eclipse is derived from the 2nd edition of Plan9. It is currently implemented on PC based machines. Table 2 describes the breakup of source lines in Plan9 and Eclipse. In both cases, we refer to an extravagant kernel including all device drivers and protocols. As Table 2 indicates, Eclipse is machine independent. The tiny machine dependent part of Eclipse is contained in the clock interrupt handler, and is shown in the Appendix. It can be easily ported to other architectures.

part	Plan9 2nd Ed.	Eclipse
portable	31,046 lines	±1,143 lines
PC dependent	19,031 lines	+22 lines
total	50,077 lines	51,040 lines

± means changed source lines (additions and deletions)

Table 2: Plan9 and Eclipse Source Size

7 Experience and Measurements

Eclipse has been operational since October 1995. It proved to be stable, useful and instructional. In particular, we can easily demonstrate and control extreme conditions, such as overloading, memory thrashing and starvation. We use Eclipse for running multimedia applications, such as MPEG players, concurrently with other demanding activities. The applications meet their deadlines when they run in reservation domains with appropriate resource reservations.

We used two methods for determining the appropriate reservations: manual trial and error (actually, a binary search), and an automatic resource monitor, which is a user-level process, that collect "complaints" from application programs and adjusts the reservations appropriately. The automatic monitor increases the reservation (fast) when the application suffers from delays, and reduces the reservation (slowly) when the application is not delayed.

We have two MPEG-1 players for Eclipse: a player which reads from a file, and a *Fellini* [9] client, which receives a stream of information from the *Fellini* continuous media server.

The rest of this section describes several experiments that illustrate the effectiveness of MTR-LS scheduling algorithm. In those experiments we used reservation domains extensively to distinguish between several classes of applications that run concurrently on the same machine.

7.1 Experimental Setup

All experiments were run on a Micron Millenia PC with an Intel Pentium processor running at 133MHz. The PC has a Talisman XL MPEG-1 decoder card.

The experiments compared three scheduling algorithms: the original Plan9 scheduler, which is a priority based scheduler, that adjusts the priorities of processes according to their CPU consumption, the MTR-LS scheduler, and a weighted round-robin (WRR) scheduler.

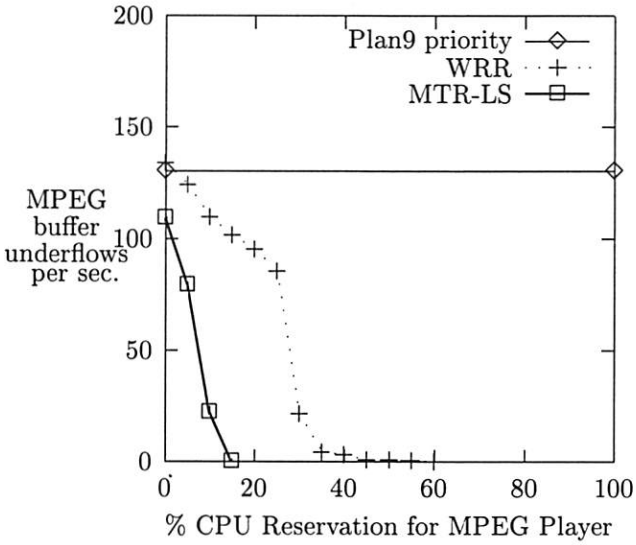


Figure 9: Buffer Underflows of an MPEG Player Under CPU Overload

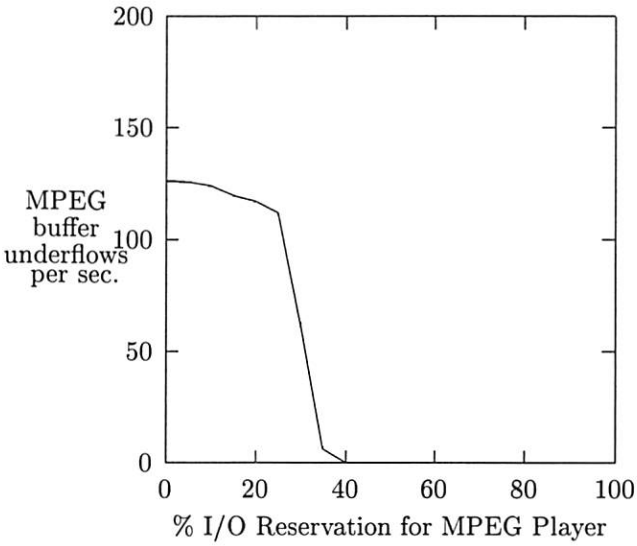


Figure 10: Buffer Underflows of an MPEG Player Under I/O Overload

The first experiment runs a MPEG player under CPU and I/O overload. The MPEG-1 player has 10 read-ahead threads, that read a movie from the local disk and send it to the MPEG decoder card. The MPEG player requires both I/O and CPU resources.

We repeatedly created CPU overload by running multiple copies of the `onoff` program. Each copy of this program consumes 11ms of CPU and then sleeps for 5ms. The `onoff` program poses a challenge to most CPU schedulers, since it consumes CPU with a duty cycle of about 60%, and it sleeps frequently.

7.2 Providing QoS in Overloaded System by Resource Reservation

Figures 9 and 10 depict the effects of resource reservation on an MPEG-1 player under CPU and I/O overloads, respectively. In the CPU load experiment, the MPEG player was run against 8 concurrent instances of the `onoff` program. In the I/O load experiment, the MPEG player was run against 10 I/O processes that were accessing random locations on the disk that holds the MPEG movie. In all cases, the file system was running in reservation domain #1 with a fixed 20% CPU reservation (where applicable). The MPEG player was running in reservation domain #2. The I/O workload was running in reservation domain #3, and each competing instance of `onoff` was running in its own reservation domain without any explicit reservations. In all cases, we varied the corresponding resource reservation of domain #2.

Figures 9 and 10 show the average number of buffer *underflows* in the MPEG card (which indicate missed deadlines). Buffer underflows are sampled at a 200Hz rate. A zero value indicates no missed deadlines, and it is always achieved when the MPEG player is running in an otherwise idle system. The value 200 indicates that all deadlines were missed (the buffer underflowed continuously).

This experiment demonstrates that *reservation domains provide guaranteed execution rate in an overloaded system.* Both MTR-LS and WRR provided good quality of service under CPU load. However, MTR-LS required a reservation of 15% to prevent underflows, while WRR required a reservation of 55%. The reason for MTR-LS superiority is that it prefers less greedy processes, such as the MPEG player, over more greedy processes, such as `onoff`, which exhausted its entire reservation. Note that each `onoff` consumes more than 60% of the CPU on an idle machine, and here it was reserved less than 9% of the CPU. The Plan9 priority scheduler failed in this experiment to prevent buffer underflows, since it did not prefer the MPEG player over the `onoff` processes. The Plan9 scheduler detects CPU bound processes, but it failed to recognize the `onoff` processes as CPU hogs since they sleep frequently.

7.3 Comparison of MTR-LS with Priority Scheduling and Weighted Round-Robin

Figure 11 compares the scheduling delays of MTR-LS with the Plan9 priority scheduler and WRR. We run a single copy of `sleeper` concurrently with a varying

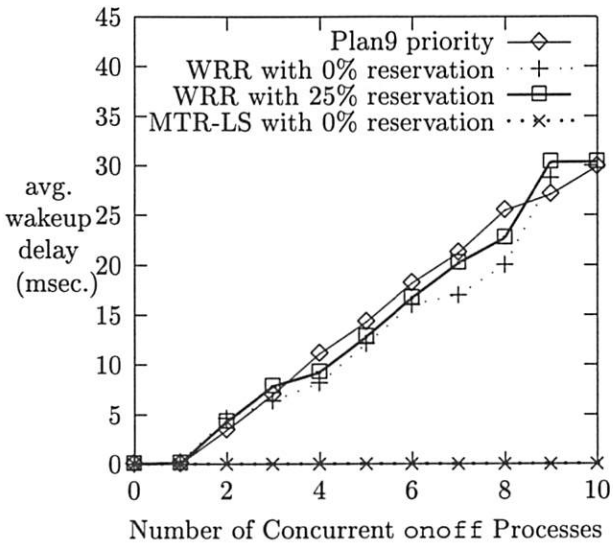


Figure 11: Wakeup Delay in Overloaded Systems

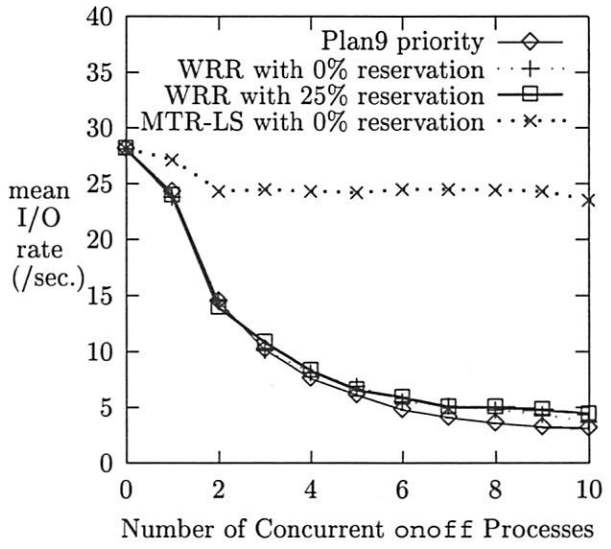


Figure 12: I/O Rate in Overloaded Systems

number of `onoff` programs under different scheduling algorithms. The `sleeper` program consumes 11msec. of CPU time and then sleeps for 100 msec. repeatedly. `Sleeper` measures the difference between its actual wakeup time and its anticipated wakeup time and reports the average delay. The `sleeper` program was run in reservation domain #2, and each instance of `onoff` was run in a separate reservation domain without any explicit CPU reservation.

Figure 11 illustrates the advantage of MTR-LS over Plan9 priority scheduling and WRR. MTR-LS always schedules the `sleeper` program before `onoff`, since `sleeper` consumes CPU at a slower rate, and thus its tokens tend to migrate to the beginning of the list $\mathcal{L}$. In other words, *MTR-LS prefers less-greedy processes*. In fact, this experiment shows that MTR-LS provides a delay bound to processes that do not exceed their reservation.

Of course, if `sleeper` had an explicit priority (`nice`) in the priority scheduler, it would not suffer from wakeup delays. However, changing the process priority may not be available in many situations, and may cause undesired side effects.

7.4 CPU Scheduling of I/O Bound Processes

Figure 12 depicts the I/O rate of a single thread of an I/O bound program when it runs concurrently with a varying number of `onoff` programs under different scheduling algorithms. The I/O bound program was run in reservation domain #2, and each instance of the `onoff` program was run in a separate domain without any explicit CPU reservation.

The most interesting result of this experiment is that the I/O rate *decreased* under increasing CPU load with the Plan9 priority scheduling and WRR. The explanation of this phenomenon is as follows: Although the I/O process has an unlimited access to the disk, it must wait for its turn for the CPU in order to issue the next I/O request. The I/O process is never granted the CPU immediately in WRR, since it is always appended to tail of the ready queue after the n `onoff` processes. This is why I/O throughput decreased as CPU contention increased. Providing CPU reservation to both the I/O process and to the file system process did not alleviate this problem. Plan9 priority scheduling failed to prefer the I/O process over the `onoff` processes, because all of them sleep frequently.

MTR-LS successfully isolated the I/O process from the CPU overload, even without any CPU reservations. The reason is that the I/O process is less CPU-greedy than the `onoff` processes. In other word, MTR-LS provided a *cumulative service* guarantee for the I/O bound process, which was not delayed by the CPU.

8 Conclusions and Future Work

In this paper we introduced a new operating system abstraction, called *reservation domains*, which can be used to provide predictable quality of service in overloaded systems and to partition resources among concurrent users. Reservation domains allow soft real-time appli-

cations to co-exist with batch applications in the same system, without any change to the applications.

We also described a new scheduling algorithm, Move-to-Rear List Scheduling (MTR-LS), that provides fairness, delay bound, and cumulative service guarantees. We explained why those QoS parameters are important. The rest of the paper described Eclipse, a new experimental operating system, that implements reservation domains and MTR-LS scheduling. Eclipse can schedule three types of resources: CPU, disk I/O and physical memory (working set). Experimental results indicate that Eclipse provide delay bounds and cumulative service bounds in overload situations. We also showed that MTR-LS prefers *less-greedy* processes, which means that it automatically provides better performance in many cases, as shown by our experiments. We compared MTR-LS with Plan9 priority scheduling and with weighted round-robin scheduling. MTR-LS provided superior results in several cases. We illustrated a simple implementation of MTR-LS in the Appendix. More time efficient implementations of MTR-LS are possible, but they use more complex data structures.

Our ongoing work concentrates on extending the reservation domain approach into a distributed environment with multiple clients and servers, investigating hierarchical scheduling for reservation domains, developing new scheduling algorithms, and porting the reservation domains and MTR-LS into more popular operating systems, such as Linux or FreeBSD.

Acknowledgments

The authors would like to thank Ron Phelps for helping with programming the experiments.

References

[1] J. Bruno, E. Gabber, B. Özden, H. Saran, and A. Silberschatz. Closed-loop packet sources and cumulative service. Technical report, Aug 1997.

[2] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing qos guarantees. In *Proceedings of ACM Multimedia, Seattle, Washington*, November 1997.

[3] A. Demers, S. Keshav, and S. Shenker. "Design and Analysis of a Fair Queuing Algorithm". In *Proceedings of the ACM SIGCOMM Austin, Texas, September 1989*, September 1989.

[4] P. Goyal, X. Guo, and H. M. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems". In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation Seattle, Washington, October 1996*, October 1996.

[5] M. B. Jones, D. Roşu, and M.-Cătălin Roşu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–211, Saint-Malo, France, October 5-8 1997.

[6] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Braham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[7] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[8] J. Niehand and M. S. Lam. The design and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–197, Saint-Malo, France, October 5-8 1997.

[9] B. Özden, R. Rastogi, and A. Silberschatz. *Fellini Continuous Media Storage Server*. Kluwer Academic Publishers.

[10] *Plan 9 Programmer's Manual, Second Edition*, volume 1 and 2. Computing Science Research Center, AT&T Bell Labs, Murray Hill, NJ, 1995. ISBN 0-03-017138-5 and 0-03-017139-3.

[11] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks–the single node case. *IEEE/ACM Transactions on Networking*, pages 344–357, June 1993.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems, The Journal of the USENIX Association*, 8(3):221–254, Summer 1995.

[13] I. Stoica and et.al. "A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems". In *Proceedings of IEEE Real-Time Systems Symposium*, December 1996.

[14] M. Thorup. "On RAM Priority Queues". In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, 1996*, pages 59–67, Atlanta, January 1996.

[15] Ian Wakeman, Atanu Ghosh, and Jon Crowcroft. Implementing real time packet forwarding policies using streams. In *Proceedings of USENIX 1995 Technical Conference*, pages 71–82, New Orleans, Louisiana, January 16-20 1995.

[16] C. A. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

[17] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating System Design and Implementation (OSDI)*, pages 1–11, Montrey, California, November 14-17 1994.

Appendix: The Eclipse Implementation of MTR-LS

```
typedef struct Proc  Proc;
typedef struct Rd   Rd;

enum {
  /* clock ticks per sched. cycle */
  CYCLE2TK = (HZ/2),
};

struct Proc {
  Proc *rnext;  /* next process in run q */
  int  state;
  Rd   *rd;     /* reservation domain */
};

struct Rd {
  Rd   *next;   /* next RD in list */
  int  rdid;    /* reservation domain ID */
        /* 0 indicates a kernel proc. */
  int  cpu_resrv; /* CPU reservation in */
        /* ticks */
  uint lo_tok;  /* lowest token time-stamp */
  /* cyclic list of time-stamps */
  uint tok_list[CYCLE2TK];
  int  head, tail;
};

/* Globals */
static Schedq runq; /* process run queue */
static RDq rd;      /* reservation domains */
uint hi_stamp;      /* current time-stamp */

/* Initialization of tokens list */
void
init_tok_list(void)
{
  Rd *r;
  int go;

  /* first token in each domain */
  hi_stamp = 0;
  for (r = rd.head; r; r = r->next) {
    r->lo_tok = r->tok_list[0] = hi_stamp++;
    r->head = r->tail = 1;
  }

  /* assign tokens in interleaved fashion */
  do {
    go = 0;
    for (r = rd.head; r; r = r->next)
      if (r->head < r->cpu_resrv) {
        go = 1;
        r->tok_list[r->head++] = hi_stamp++;
      }
  } while(go);
}

/* Clock interrupt routine */
static void
clock(Ureg *ur, void *arg)
{
  Proc *p;
  Rd *r;

  p = m->proc;
  if (p) {
    if (p->state == Running) {
      r = p->rd;
      if (++hi_stamp == 0)
        /* re-initialize tokens list at */
        /* time-stamp overflow */
        init_tok_list();
      r->tok_list[r->head++] = hi_stamp;
      if (r->head >= CYCLE2TK)
        r->head = 0;
      r->lo_tok = r->tok_list[r->tail++];
      if (r->tail >= CYCLE2TK)
        r->tail = 0;
    }
  }

  if (u && p && p->state == Running) {
    /* preempt if not holding a spin lock */
    ...
    sched();
  }
}

/* Select the next process to run */
Proc*
runproc(void)
{
  Proc *p, *prev, *best, *prevb;

loop:
  prev = best = 0;
  for (p = runq.head; p; p = p->rnext) {
    /* run kernel processes first */
    if (p->rd->rdid == 0) {
      best = p;
      prevb = prev;
      break;
    }

    if (best == 0 ||
        p->rd->lo_tok < best->rd->lo_tok) {
      best = p;
      prevb = prev;
    }
    prev = p;
  }

  if (best == 0)
    goto loop;

  /* remove process from run queue */
  remq(&runq, best, prevb);
  return best;
}
```

A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers

Kenjiro Cho

Sony Computer Science Laboratory, Inc.
Tokyo, Japan 1410022
kjc@csl.sony.co.jp

Abstract

Queueing is an essential element of traffic management, but the only queueing discipline used in traditional UNIX systems is simple FIFO queueing. This paper describes ALTQ, a queueing framework that allows the use of a set of queueing disciplines. We investigate the issues involved in designing a generic queueing framework as well as the issues involved in implementing various queueing disciplines. ALTQ is implemented as simple extension to the FreeBSD kernel including minor fixes to the device drivers. Several queueing disciplines including CBQ, RED, and WFQ are implemented onto the framework to demonstrate the design of ALTQ. The traffic management performance of a PC is presented to show the feasibility of traffic management by PC-UNIX based routers.

1 Introduction

Traffic management is of great importance to today's packet networks. Traffic management consists of a diverse set of mechanisms and policies, but the heart of the technology is a packet scheduling mechanism, also known as queueing. Sophisticated queueing can provide performance bounds of bandwidth, delay, jitter, and loss, and thus, can meet the requirements of real-time services. Queueing is also vital to best-effort services to avoid congestion and to provide fairness and protection, which leads to more stable and predictable network behavior. There has been a considerable amount of research related to traffic management and queueing over the last several years.

Many queueing disciplines have been proposed and studied to date by the research community, mostly by analysis and simulation. Such disciplines are not, however, widely used because there is no easy way to implement them into the existing network equipment. In BSD UNIX, the only queueing discipline implemented is a simple tail-drop FIFO queue. There is no general method to implement an alternative queueing discipline, which is the main obstacle to incorporating alternative queueing disciplines.

On the other hand, the rapidly increasing power of PCs, emerging high-speed network cards, and their dropping costs make it an attractive choice to implement an intelligent queueing on PC-based routers. Another driving force behind PC-based routers is flexibility in software development as the requirements for a router are growing.

In view of this situation, we have designed and built ALTQ, a framework for alternate queueing. ALTQ allows implementors to implement various queueing disciplines on PC-based UNIX systems. A set of queueing disciplines are implemented to demonstrate the traffic management abilities of PC-UNIX based routers.

ALTQ is designed to support a variety of queueing disciplines with different components: scheduling strategies, packet drop strategies, buffer allocation strategies, multiple priority levels, and non-work conserving queues. Different queueing disciplines can share many parts: flow classification, packet handling, and device driver support. Therefore, researchers will be able to implement a new queueing discipline without knowing the details of the kernel implementations.

Our framework is designed to support both research and operation. Once such a framework is widely deployed, research output by simulation can be easily implemented and tested with real machines and networks. Then, if proved useful, the discipline can be brought into routers in practical service. Availability of a set of queueing disciplines will raise public awareness of traffic management issues, which in turn raises research incentives to attack hard problems.

Another important issue to consider is deployment of the framework. To this end just a framework is not enough. In order to make people want to use it, we have to have concrete queueing disciplines which have specific applications. Therefore, we have implemented

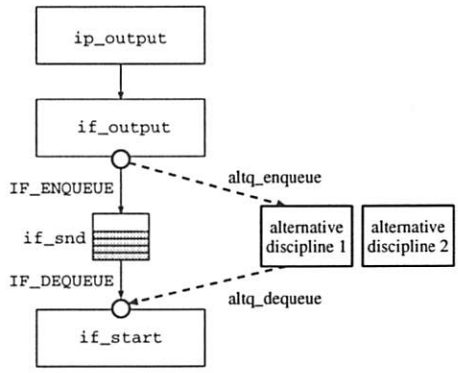


Figure 1: Alternate Queueing Architecture

Class-Based Queueing (CBQ) [6] and Random Early Detection (RED) [5] targeting two potential user groups. CBQ can be used for integrated services. RSVP [22] is a resource reservation protocol for integrated services; RSVP itself is a signaling protocol to set up the traffic control module of the routers along a path. The RSVP release from ISI [9] does not have a traffic control module so that there are great demands for a queueing implementation capable of traffic control. Although CBQ was not originally designed for use with RSVP, CBQ has been used by Sun as a traffic control module for RSVP on Solaris and the source code has been available [20].

RED is for active queue management of best-effort services [2]. Active queue management has been extensively discussed to avoid congestion in the Internet. Although CBQ can be used for this purpose, RED seems more popular since RED does not require flow states or class states, and thus, is simpler and more scalable. Again, no implementation is available at hand.

Another important factor for widespread acceptance is simplicity and stability of the implementation, which caused us to emphasize practicality instead of elegance while designing ALTQ.

In summary, the goals of our prototype are three-fold:

- provide a queueing research platform.
- provide a traffic control kernel for RSVP.
- make active queue management available.

In this paper, we will show the design and implementation of ALTQ, and report the traffic management performance of the prototype on FreeBSD [7].

2 ALTQ

2.1 ALTQ Design

The basic design of ALTQ is quite simple; the queueing interface is a switch to a set of queueing disciplines as shown in Figure 1. Alternate queueing is used only

```
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);
    splx(s);
    m_freem(m);
    return(ENOBUFS);
}
IF_ENQUEUE(&ifp->if_snd, m);
if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);
splx(s);
```

Figure 2: Enqueue Operation in *if_output*

for the output queue of a network interface. The input queue has less importance because traffic control functions only at the entrance to a bottleneck link. In BSD UNIX, an output queue is implemented in the abstract interface structure *ifnet*. The queue structure, *if_snd*, is manipulated by *IF_ENQUEUE()* and *IF_DEQUEUE()* macros. These macros are used by two functions registered in *struct ifnet*, *if_output* and *if_start*; *if_output* defined for each link type performs the enqueue operation, and *if_start* defined as part of a network device driver performs the dequeue operation [13].

One might think that just replacing *IF_ENQUEUE()* and *IF_DEQUEUE()* will suffice, but, unfortunately, it is not the case. The problem is that the queueing operations used inside the kernel are not only enqueueing and dequeueing. In addition, surprisingly many parts of the kernel code assume FIFO queueing and the *ifqueue* structure.

To illustrate the problem, let's take a look at the enqueue operation in a typical *if_output* in Figure 2. The code performs three operations related to queueing.

1. check if the queue is full by *IF_QFULL()*, and if so, drop the arriving packet.
2. enqueue the packet by *IF_ENQUEUE()*.
3. call the device driver to send out the packet unless the driver is already busy.

The code assumes the tail-drop policy, that is, the arriving packet is dropped. But the decision to drop and the selection of a victim packet should be up to a queueing discipline. Moreover, in a random drop policy, the drop operation often comes after enqueueing an arriving packet. The order of the two operations also depends on a queueing discipline. Furthermore, in a non-work conserving queue, enqueueing a packet does not mean the packet is sent out immediately, but rather, the driver should be invoked later at some scheduled timing. Hence, in order to implement a generic queueing interface, we have no choice but to replace part of the code in *if_output* routines.

There are also problems in *if_start* routines. Some drivers peek at the head of the queue to see if the driver

has enough buffer space and/or DMA descriptors for the next packet. Those drivers directly access *if_snd* using different methods since no procedure is defined for a peek operation. A queueing discipline could have multiple queues, or could be about to dequeue a packet other than the one at the head of the queue. Therefore, the peek operation should be part of the generic queueing interface. Although it is possible in theory to rewrite all drivers not to use peek operations, it is wise to support a peek operation, considering the labor required to modify the existing drivers. A discipline must guarantee that the peeked packet will be returned by the next dequeue operation.

IF_PREPEND() is defined in BSD UNIX to add a packet at the head of the queue, but the prepend operation is intended for a FIFO queue and should not be used for a generic queueing interface. Fortunately, the prepend operation is rarely used—with the exception of one popular Ethernet driver of FreeBSD. This driver uses *IF_PREPEND()* when there are not enough DMA descriptors available, to put back a dequeued packet. We had to modify this driver to use a peek-and-dequeue method instead.

Another problem in *if_start* routines is a queue flush operation to empty the queue. Since a non-work conserving queue cannot be emptied by a dequeue loop, the flush operation should be defined.

In summary, the requirements of a queueing framework to support various queueing disciplines are:

- a queueing framework should support enqueue, dequeue, peek, and flush operations.
- an enqueue operation is responsible for dropping packets and starting drivers.
- drivers may use a peek operation, but should not use a prepend operation.

2.2 ALTQ Implementation

Our design policy is to make minimal changes to the existing kernel, but it turns out that we have to modify both *if_output* routines and *if_start* routines because the current queue operations do not have enough abstraction. Modifying *if_start* means modifications to drivers and it is not easy to modify all the existing drivers. Therefore, we took an approach that allows both modified drivers and unmodified drivers to coexist so that we can modify only the drivers we need, and incrementally add supported drivers. This is done by leaving the original queueing structures and the original queueing code intact, and adding a hook to switch to alternate queueing. By doing this, the kernel, unless alternate queueing is enabled, follows the same sequence using the same references as in the original—with the exception of test of the hook. This method has other advantages; the system can fall back to the original queueing if something goes wrong. As a result, the system becomes more reliable, easier to use, and easier to debug. In addition, it is compatible with the existing user programs that refer to *struct ifnet* (e.g., ifconfig and netstat).

Queueing disciplines are controlled by *ioctl* system calls via a queueing device (e.g., */dev/cbq*). ALTQ is defined as a character device and each queueing discipline is defined as a minor device of ALTQ. To activate an alternative queueing discipline, a privileged user program opens the queue device associated with the discipline, then, attaches the discipline to an interface and enables it via the corresponding *ioctl* system calls. When the alternative queueing is disabled or closed, the system falls back to the original FIFO queueing.

Several fields are added to *struct ifnet* since *struct ifnet* holds the original queue structures, and is suitable to place the alternate queueing fields. The added fields are a discipline type, a common state field, a pointer to a discipline specific state, and pointers to discipline specific enqueue/dequeue functions.

Throughout the modifications to the kernel for ALTQ, the *ALTQ_IS_ON()* macro checks the ALTQ state field in *struct ifnet* to see if alternate queueing is currently used. When alternate queueing is not used, the original FIFO queueing code is executed. Otherwise, the alternative queue operations are executed.

Two modifications are made to *if_output* routines. One is to pass the protocol header information to the enqueue operation. The protocol header information consists of the address family of a packet and a pointer to the network layer header in the packet. Packet classifiers can use this information to efficiently extract the necessary fields from a packet header. Packet marking can also be implemented using the protocol header information. Alternatively, flow information can be extracted in the network layer, or in queueing operations without the protocol header information. However, if flow information extraction is implemented in the network layer, the *if_output* interface should be changed to pass the information to *if_output* or auxiliary data should be added to *mbuf* structure, which affects fairly large part of the kernel code. On the other hand, if flow information extraction is implemented entirely in enqueue operations, it has to handle various link-level headers to locate the network layer header. Since we have to modify *if_output* routines anyway to support the enqueue operation of ALTQ, our choice is to save the minimum information in *if_output* before prepending the link header and pass the protocol header information to the enqueue operation.

The second modification to *if_output* routines is to support the ALTQ enqueue operation as shown in Figure 3. The ALTQ enqueue function is also responsible for drop-

```
    s = splimp();
#ifdef ALTQ
    if (ALTQ_IS_ON(ifp)) {
        error = (*ifp->if_altqenqueue)(ifp, m,
                        &pr_hdr, ALTEQ_NORMAL);
        if (error) {
            splx(s);
            return (error);
        }
    }
    else {
#endif
        if (IF_QFULL(&ifp->if_snd)) {
            IF_DROP(&ifp->if_snd);
            splx(s);
            m_freem(m);
            return(ENOBUFS);
        }
        IF_ENQUEUE(&ifp->if_snd, m);
        if ((ifp->if_flags & IFF_OACTIVE) == 0)
            (*ifp->if_start)(ifp);
#ifdef ALTQ
    }
#endif
    splx(s);
```

Figure 3: Modified Enqueue Operation in *if_output*

```
#ifdef ALTQ
    if (ALTQ_IS_ON(ifp))
        m = (*ifp->if_altqdequeue)(ifp,
                                ALTDQ_DEQUEUE);
    else
#endif
        IF_DEQUEUE(&ifp->if_snd, m);
```

Figure 4: Modified Dequeue Operation in *if_start*

ping packets and starting the driver.

Similarly, *if_start* routines are modified to use alternate queueing as shown in Figure 4. A peek operation and a flush operation can be done by calling a dequeue routine with *ALTDQ_PEEK* or *ALTDQ_FLUSH* as a second parameter. Network device drivers modified to support ALTQ can be identified by setting the *ALTQF_READY* bit of the ALTQ state field in *struct ifnet*. This bit is checked when a discipline is attached.

3 Queueing Disciplines

3.1 Overview of Implemented Disciplines

First, we briefly review the implemented disciplines. The details of the mechanisms, simulation results, and analysis can be found elsewhere [6, 21, 5, 14, 3, 11, 12].

CBQ (Class-Based Queueing)

CBQ was proposed by Jacobson and has been studied by Floyd [6]. CBQ has given careful consideration to implementation issues, and is implemented as a STREAMS module by Sun, UCL and LBNL [21]. Our CBQ code is ported from CBQ version 2.0 and enhanced.

CBQ achieves both partitioning and sharing of link bandwidth by hierarchically structured classes. Each class has its own queue and is assigned its share of bandwidth. A child class can borrow bandwidth from its parent class as long as excess bandwidth is available.

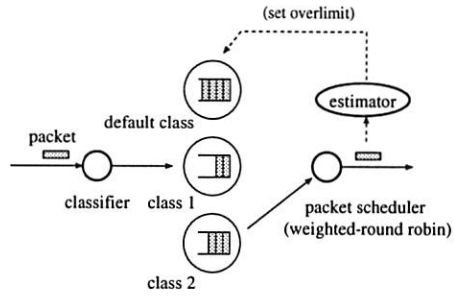


Figure 5: CBQ Components

Figure 5 shows the basic components of CBQ. CBQ works as follows: The classifier assigns arriving packets to the appropriate class. The estimator estimates the bandwidth recently used by a class. If a class has exceeded its predefined limit, the estimator marks the class as overlimit. The scheduler determines the next packet to be sent from the various classes, based on priorities and states of the classes. Weighted-round robin scheduling is used between classes with the same priority.

RED (Random Early Detection)

RED was also introduced by Floyd and Jacobson [5]. RED is an implicit congestion notification mechanism that exercises packet dropping or packet marking stochastically according to the average queue length. Since RED does not require per-flow state, it is considered scalable and suitable for backbone routers. At the same time, RED can be viewed as a buffer management mechanism and can be integrated into other packet scheduling schemes.

Our implementation of RED is derived from the RED module in the NS simulator version 2.0. Explicit Congestion Notification (ECN) [4], a packet marking mechanism under standardization process, is experimentally supported. RED and ECN are integrated into CBQ so that RED and ECN can be enabled on a class queue basis.

WFQ (Weighted-Fair Queueing)

WFQ [14, 3, 11] is the best known and the best studied queueing discipline. In a broad sense, WFQ is a discipline that assigns a queue for each flow. A weight can be assigned to each queue to give a different proportion of the network capacity. As a result, WFQ can provide protection against other flows. In the queueing research community, WFQ is more precisely defined as the specific scheduling mechanism proposed by Demers et al. [3] that is proved to be able to provide worst-case end-to-

end delay bounds [15]. Our implementation is not WFQ in this sense, but is closer to a variant of WFQ, known as SFQ or stochastic fairness queueing [12]. A hash function is used to map a flow to one of a set of queues, and thus, it is possible for two different flows to be mapped into the same queue. In contrast to WFQ, no guarantee can be provided by SFQ.

FIFOQ (First-In First-Out Queueing)

FIFOQ is nothing but a simple tail-drop FIFO queue that is implemented as a template for those who want to write their own queueing disciplines.

3.2 Implementation Issues

There are issues and limitations which are generic when porting a queueing discipline to the ALTQ framework. We discuss these issues in this section, but details specific to particular queueing disciplines are beyond the scope of this paper.

Implementing a New Discipline

We assume that a queueing discipline is evaluated by simulation, and then ported onto ALTQ. The NS simulator [18] is one of a few simulators that support different queueing disciplines. The NS simulator is widely used in the research community and includes the RED and CBQ modules.

To implement a new queueing discipline in ALTQ, one can concentrate on the enqueue and dequeue routines of the new discipline. The FIFOQ implementation is provided as a template so that the FIFOQ code can be modified to put a new queueing discipline into the ALTQ framework. The basic steps are just to add an entry to the ALTQ device table, and then provide open, close, and ioctl routines. The required *ioctls* are attach, detach, enable, and disable. Once the above steps are finished, the new discipline is available on all the interface cards supported by ALTQ.

To use the added discipline, a privileged user program is required. Again, a daemon program for FIFOQ included in the release should serve as a template.

Heuristic Algorithms

Queueing algorithms often employ heuristic algorithms to approximate the ideal model for efficient implementation. But sometimes properties of these heuristics are not well studied. As a result, it becomes difficult to verify the algorithm after it is ported into the kernel.

The *Top-Level link-sharing* algorithm of CBQ suggested by Floyd [6] is an example of such an algorithm. The algorithm employs heuristics to control how far the scheduler needs to traverse the class tree. The suggested heuristics work fine with their simulation settings, but do not work so well under some conditions. It requires time-consuming efforts to tune parameters by heuristics. Although good heuristics are important for efficient implementation, heuristics should be carefully used and study of properties of the employed heuristics will be a great help for implementors.

Blocking Interrupts

Interrupts should be blocked when manipulating data structures shared with the dequeue operation. Dequeue operations are called in the device interrupt level so that the shared structures should be guarded by blocking interrupts to avoid race conditions. Interrupts are blocked during the execution of *if_start* by the caller.

Precision of Integer Calculation

32-bit integer calculations easily overflow or underflow with link bandwidth varying from 9600bps modems to 155Mbps ATM. In simulators, 64-bit double precision floating-point is available and it is reasonable to use it to avoid precision errors. However, floating-point calculation is not available or not very efficient in the kernel since the floating-point registers are not saved for the kernel (in order to reduce overhead). Hence, algorithms often need to be converted to use integers or fixed-point values. Our RED implementation uses fixed-point calculations converted from floating-point calculations in the NS simulator. We recommend performing calculations in the user space using floating-point values, and then bringing the results into the kernel. CBQ uses this technique. The situation will be improved when 64-bit integers become more commonly used and efficient.

Knowing Transfer Completion

Queueing disciplines may need to know the time when a packet transmission is completed. CBQ is one of such disciplines. In BSD UNIX, the *if_done* entry point is provided as a callback function for use when the output queue is emptied [13], but no driver supports this callback. In any case, a discipline needs to be notified when each packet is transferred rather than when the queue is emptied. Another possible way to know transfer completion is to use the callback hook of a memory buffer (e.g., *mbuf cluster*) so that the discipline is notified when the buffer is freed. The CBQ release for Solaris uses this technique. The problem with this method is that the callback is executed when data finishes transferring to the interface, rather than to the wire. Putting a packet on the wire takes much longer than DMAing the packet to the interface. Our CBQ implementation does not use these callbacks, but estimates the completion time from the packet size when a packet is queued. Though this is not an ideal solution, it is driver-independent and provides estimates good enough for CBQ.

Time Measurements

One should be aware of the resolution and overhead of getting the time value. Queueing disciplines often need to measure time to control packet scheduling. To get wall clock time in the kernel, BSD UNIX provides a *microtime()* call that returns the time offset from 1970 in microseconds. Intel Pentium or better processor has a 64-bit time stamp counter driven by the processor clock, and this counter can be read by a single instruction. If the processor clock is 200MHz, the resolution is 5 nanoseconds. The PC based UNIX systems use this counter for *microtime()* when available. Alternatively, one can directly read the time stamp counter for efficiency or for high resolution. Even better, the counter value is never adjusted as opposed to *microtime()*. The problem is that processors have different clocks so that the time stamp counter value needs to be normalized to be usable on different machines. Normalization requires expensive multiplications and divisions, and the low order bits are subject to rounding errors. Thus, one should be careful about precision and rounding errors. Since *microtime()* requires only a microsecond resolution, it is coded to normalize the counter value by a single multiplication and to have enough precision. *Microtime()* takes about 450 nanoseconds on a PentiumPro 200MHz machine. The ALTQ implementation currently uses *microtime()* only.

Timer Granularity

Timers are frequently used to set timeout-process routines. While timers in a simulator have almost infinite precision, timers inside the kernel are implemented by an interval timer and have limited granularity. Timer granularity and packet size are fundamental factors to packet scheduling.

To take one example, the accuracy of the bandwidth control in CBQ relies on timer granularity in the following way: CBQ measures the recent bandwidth use of each class by averaging packet intervals. CBQ regulates a class by suspending the class when the class exceeds its limit. To resume a suspended class, CBQ needs a trigger, either a timer event or a packet input/output event. In the worst case scenario where there is no packet event, resume timing is rounded up to the timer granularity. Most UNIX systems use 10 msec timer granularity as default, and CBQ uses 20 msec as the minimum timer.

Each class has a variable *maxburst* and can send at most *maxburst* back-to-back packets. If a class sends *maxburst* back-to-back packets at the beginning of a 20 msec cycle, the class gets suspended and would not be resumed until the next timer event—unless other event triggers occur. If this situation continues, the transfer rate becomes

$$rate = packetsize \times maxburst \times 8 \div 0.02$$

Now, assume that *maxburst* is 16 (default) and the packet size is the link MTU. For 10baseT with a 1500-byte MTU, the calculated rate is 9.6Mbps. For ATM with a 9180-byte MTU, the calculated rate is 58.8Mbps.

A problem arises with 100baseT; it is 10 times faster than 10baseT, but the calculated rate remains the same as 10baseT. CBQ can fill only 1/10 of the link bandwidth. This is a generic problem in high-speed network when packet size is small compared to the available bandwidth. Because increasing *maxburst* or the packet size by a factor of 10 is problematic, a fine-grained kernel timer is required to handle 100baseT. Current PCs seem to have little overhead even if timer granularity is increased by a factor of 10. The problem with 100baseT and the effect of a fine-grained timer are illustrated in Section 4.3.

Depending solely on the kernel timer is, however, the worst case. In more realistic settings, there are other flows or TCP ACKs that can trigger CBQ to calibrate sending rates of classes.

Slow Device with Large Buffer

Some network devices have large buffers and a large send buffer adversely affects queueing. Although a large receive buffer helps avoid overflow, a large send buffer just spoils the effect of intelligent queueing, especially when the link is slow. For example, if a device for a 128Kbps link has a 16KB buffer, the buffer can hold 1 second worth of packets, and this buffer is beyond the control of queueing. The problem is invisible under FIFO queueing. However, when better queueing is available, the send buffer size of a device should be set to the minimum amount that is required to fill up the pipe.

4 Performance

In this section, we present the performance of the implemented disciplines. Note that sophisticated queueing becomes more important at a bottleneck link, and thus, its performance does not necessarily correspond to the high-speed portion of a network.

We use primarily CBQ to illustrate the traffic management performance of PC-UNIX routers since CBQ is the most complex and interesting of the implemented disciplines. That is, CBQ is non-work conserving, needs a classifier, and uses the combination scheduling of priority and weighted-round robin. However, we do not attempt to outline the details specific to CBQ.

4.1 Test System Configuration

We have measured the performance using three PentiumPro machines (all 200MHz with 440FX chipset) running FreeBSD-2.2.5/altq-1.0.1. Figure 6 shows the test system configuration. Host A is a source, host B is a router, and host C is a sink. CBQ is enabled only

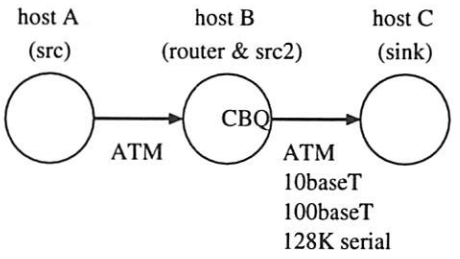


Figure 6: Test System Configuration

on the interface of host B connected to host C. The link between host A and host B is 155Mbps ATM. The link between host B and host C is either 155M ATM, 10baseT, 100baseT, or 128K serial line. When 10baseT is used, a dumb hub is inserted. When 100baseT is used, a direct connection is made by a cross cable, and the interfaces are set to the full-duplex mode. Efficient Network Inc. ENI-155p cards are used for ATM, Intel EtherExpress Pro/100B cards are used for 10baseT and 100baseT. RISCom/N2 cards are used for a synchronous serial line.

The Netperf benchmark program [10] is used with ±2.5% confidence interval at 99% confidence level. We use TCP to measure the packet forwarding performance under heavy load. In contrast to UDP, which consumes CPU cycles to keep dropping excess packets, TCP quickly adapts to the available bandwidth, and thus does not waste CPU cycle. However, a suitable window size should be selected carefully according to the end-to-end latency and the number of packets queued inside the network. Also, one should be careful about traffic in the reverse direction, since ACKs play a vital role in TCP. Especially with shared media (e.g., Ethernet), sending packets could choke TCP ACKs.

4.2 CBQ Overhead

The overhead introduced by CBQ consists of four steps: (1) extract flow information from an arriving packet. (2) classify the packet to the appropriate class. (3) select an eligible class for sending next. (4) estimate bandwidth use of the class and maintain the state of the class. There are many factors which affect the overhead: structure of class hierarchy, priority distribution, number of classes, number of active classes, rate of packet arrival, distribution of arrival, and so on. Hence, the following measurements are not intended to be complete.

Throughput Overhead

Table 1 compares TCP throughput of CBQ with that of the original FIFO queueing, measured over different link types. A small CBQ configuration with three classes is used to show the minimum overhead of CBQ. There is no other background traffic during the measurement.

Table 1: CBQ Throughput

Link Type	orig. FIFO (Mbps)	CBQ (Mbps)	overhead (%)
ATM	132.98	132.77	0.16
10baseT	6.52	6.45	1.07
100baseT	93.11	92.74	0.40
loopback			
MTU 16384	366.20	334.77	8.58
MTU 9180	337.96	314.04	7.08
MTU 1500	239.21	185.07	22.63

Table 2: CBQ Latency

Link Type	queue type	request/ response (bytes)	trans. per sec	calc'd RTT (usec)	diff (usec)
ATM	FIFO	1, 1	2875.20	347.8	
	CBQ		2792.87	358.1	10.3
	FIFO	64,64	2367.90	422.3	
	CBQ		2306.93	433.5	11.2
	FIFO	1024,64	1581.16	632.4	
	CBQ		1552.03	644.3	11.9
	FIFO	8192,64	434.61	2300.9	
	CBQ		432.64	2311.1	10.2
10baseT	FIFO	1,1	2322.40	430.6	
	CBQ		2268.17	440.9	10.3
	FIFO	64, 64	1813.52	551.4	
	CBQ		1784.32	560.4	9.0
	FIFO	1024,64	697.97	1432.7	
	CBQ		692.76	1443.5	10.8

No significant CBQ overhead is observed from the table because CBQ packet processing can overlap the sending time of the previous packet. As a result, use of CBQ does not affect the throughput.

The measurements over the software loopback interface with various MTU sizes are also listed in the table. These values show the limit of the processing power and the CBQ overhead in terms of CPU cycle. CBQ does have about 7% overhead with 9180-byte MTU, and about 23% overhead with 1500-byte MTU. It also shows that a current PC can handle more than 300Mbps with bi-directional loopback load. That is, a PC-based router has processing power enough to handle multiple 100Mbps-class interfaces; CPU load will be much lower with physical interfaces since DMA can be used. On a 300MHz PentiumII machine, we observed the loopback throughput of 420.62Mbps with 16384-byte MTU.

Latency Overhead

Table 2 shows the CBQ overhead in latency over ATM and 10baseT. In this test, request/reply style transactions are performed using UDP, and the test measures how many transactions can be performed per second. The rightmost two columns show the calculated average round-trip time (RTT) and the difference in microseconds. Again, CBQ has three classes, and there is no background traffic. We see from the table that the increase of RTT by CBQ is almost constant regardless of

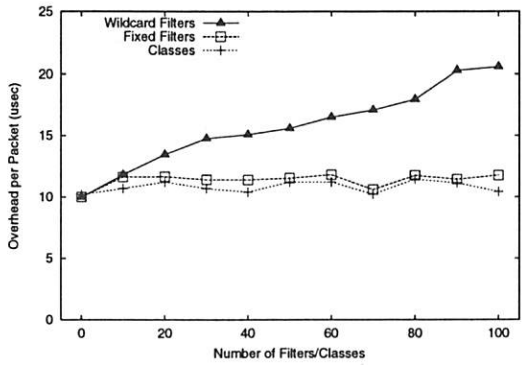


Figure 7: Effect of Number of Filters/Classes

packet size or link type, that is, the CBQ overhead per packet is about 10 microseconds.

Scalability Issues

CBQ is designed such that a class tree has relatively small number of classes; a typical class tree would have less than 20 classes. Still, it is important to identify the scalability issues of CBQ. Although a full test of scalability is difficult, the following measurements provide some insight into it. Figure 7 shows how the latency changes when we add additional filters or classes; up to 100 filters or classes are added. The values are differences in calculated RTT from the original FIFO queueing measured over ATM with 64-byte request and 64-byte response.

The "Wildcard Filters" plot and "Fixed Filters" plot in the graph show the effect of two different types of filters. To classify a packet, the classifier performs filter-matching by comparing packet header fields (e.g., IP addresses and port numbers) for each packet. In our implementation, class filters are hashed by the destination addresses in order to reduce the number of filter matching operations. However, if a filter doesn't specify a destination address, the filter is put onto the wildcard-filter list. When classifying a packet, the classifier tries the hashed list first, and if no matching is found, it tries the wildcard list. In this implementation, per-packet overhead grows linearly with the number of wildcard filters. A classifier could be implemented more efficiently, for example, using a directed acyclic graph (DAG) [1].

On the other hand, the number of classes doesn't directly affect the packet scheduler. As long as classes are underlimit, the scheduler can select the next class without checking the states of the other classes. However, to schedule a class which exceeds its share, the scheduler should see if there is a class to be scheduled first. Note that because the maximum number of overlimit classes is bound by the link speed and the minimum packet size, the overhead will not grow beyond a certain point.

When there are overlimit classes, it is obvious that CBQ performs much better than FIFO. We do not have

Table 3: Queueing Overhead Comparison

	FIFO	FIFOQ	RED	WFQ	CBQ	CBQ +RED
(usec)	0.0	0.14	1.62	1.95	10.72	11.97

numbers for such a scenario because it is difficult in our test configuration to separate the CBQ overhead from other factors (e.g., the overhead at the source and the destination hosts). But the dominant factor in latency will be the device level buffer. The measured latency will oscillate due to head-of-line blocking. The CBQ overhead itself will be by an order of magnitude smaller.

Overhead of Other Disciplines

The latency overhead can be used to compare the minimum overhead of the implemented disciplines. Table 3 shows the per-packet latency overhead of the implemented disciplines measured over ATM with 64-byte request and 64-byte response. The values are differences in calculated RTT from the original FIFO queueing. The difference of the original FIFO and our FIFOQ is that the enqueue and dequeue operations are macros in the original FIFO but they are function calls in ALTQ.

Impact of Latency Overhead

Network engineers seem to be reluctant to put extra processing on the packet forwarding path. But when we talk about the added latency, we should also take queueing delay into consideration. For example, a 1KB packet takes 800 microseconds to be put onto a 10Mbps link. If two packets are already in the queue, an arriving packet could be delayed more than 1 millisecond. If the dominant factor of the end-to-end latency is queueing delay, sophisticated queueing is worth it.

4.3 Bandwidth Allocation

Figure 8, 9 and 10 shows the accuracy of bandwidth allocation over different link types. TCP throughputs were measured when a class is allocated 5% to 95% of the link bandwidth. The plot of 100% shows the throughput when the class can borrow bandwidth from the root class. As the graphs show, the allocated bandwidth changes almost linearly over ATM, 10baseT and a serial line. However, considerable deviation is observed over 100baseT, especially during the range from 15% to 55%.

The problem in the 100baseT case is the timer granularity problem described in Section 3.2. The calculated limit rate is 9.6Mbps, and the throughput in the graph stays at this limit up to 55%. Then, as the sending rate increases, packet events help CBQ scale beyond the limit. To back up this theory, we tested the performance of the kernel whose timer granularity is modified from 10ms to 1ms. With this kernel, the calculated limit rate is 96Mbps. The result, shown as *100baseT-1KHzTimer*,

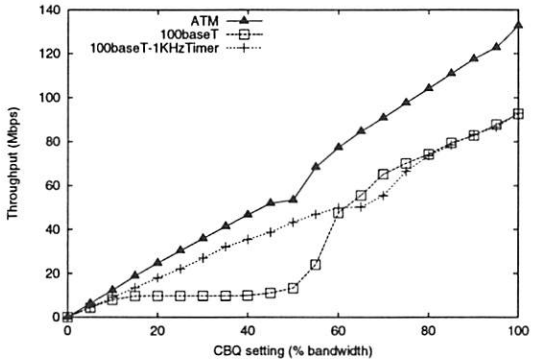


Figure 8: Bandwidth Allocation over ATM/100baseT

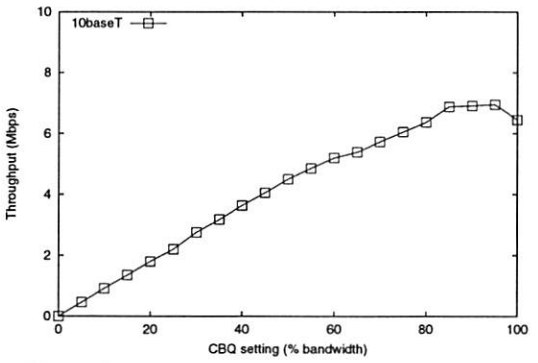


Figure 9: Bandwidth Allocation over 10baseT

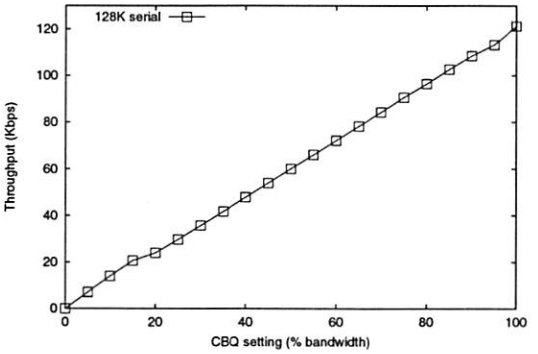


Figure 10: Bandwidth Allocation over 128K serial

is satisfactory, which also agrees with theory. Note that the calculated limit of the ATM case is 58.8Mbps, and we can observe a slight deviation at 50%, but packet events help CBQ scale beyond the limit. Also, note that 10baseT shows saturation of shared-media, and performance peaks at 85%. The performance of 10baseT drops when we try to fill up the link.

4.4 Bandwidth Guarantee

Figure 11 illustrates the success of bandwidth guarantee over ATM. Four classes, one each allocated 10Mbps, 20Mbps, 30Mbps and 40Mbps, are defined. A background TCP flow matching the default class is sent dur-

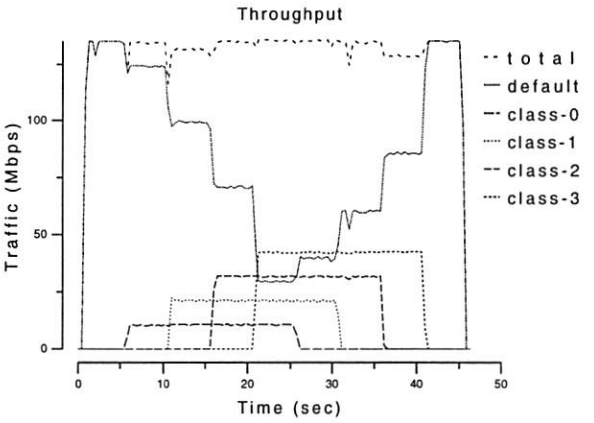


Figure 11: CBQ Bandwidth Guarantee

ing the test period. Four 20-second-long TCP flows, each corresponding to the defined classes, start 5 seconds apart from each other. To avoid oscillation caused by process scheduling, class-0 and class-2 are sent from host B and the other three classes are sent from host A. All TCP connections are trying to fill up the pipe, but the sending rate is controlled by CBQ at host B.

The *cbqprobe* tool is used to obtain the CBQ statistics (total number of octets sent by a class) every 400 msec via *ioctl*, and the *cbqmonitor* tool is used to make the graph. Both tools are included in the release.

As we can see from the graph, each class receives its share and there is no interference from other traffic. Also note that the background flow receives the remaining bandwidth, and the link is almost fully utilized during the measurement.

4.5 Link Sharing by Borrowing

Link sharing is the ability to correctly distribute available bandwidth in a hierarchical class tree. Link-sharing allows multiple organizations or multiple protocols to share the link bandwidth and to distribute "excess" bandwidth according to the class tree structure. Link-sharing has a wide range of practical applications. For example, organizations sharing a link can receive the available bandwidth proportional to their share of the cost. Another example is to control the bandwidth use of different traffic types, such as telnet, ftp, or real-time video.

The test configuration is similar to the two agency setting used by Floyd [6]. The class hierarchy is defined as shown in Figure 12 where two agencies share the link, and interactive and non-interactive leaf classes share the bandwidth of each agency. In the measurements, Agency X is emulated by host B and agency Y is emulated by host A. Four TCP flows are generated as in Figure 13. Each TCP tries to send at its maximum rate, except for the idle period. Each agency should receive its share of bandwidth all the time even when one of the leaf classes is idle, that is, the sum of class-0 and class-1 and the sum

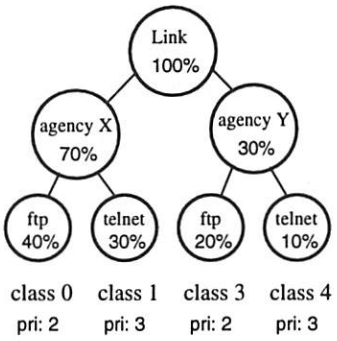


Figure 12: Class Configuration

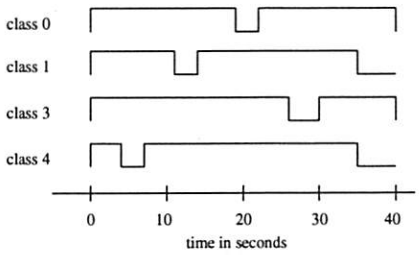


Figure 13: Test Scenario

of class-3 and class-4 should be constant.

Figure 14 shows the traffic trace generated by the same method described for Figure 11. The classes receive their share of the link bandwidth and, most of the time, receive the "excess" bandwidth when the other class in the same agency is idle. High priority class-4, however, receives more than its share in some situations (e.g., time frame:22–25). The combination of priority and borrowing in the current CBQ algorithm, especially when a class has a high priority but a small share of bandwidth, does not work so well as in the NS simulator [6]. To confirm the cause of the problem, we tested with all the classes set to the same priority. As Figure 15 shows, the problem of class-4 is improved. Note that, even if interactive and non-interactive classes have the same priority, interactive classes are likely to have much shorter latency because interactive classes are likely to have much fewer packets in their queues.

5 Discussion

One of our goals is to promote the widespread use of UNIX-based routers. Traffic management is becoming increasingly important, especially at network boundaries that are points of congestion. Technical innovations are required to provide smoother and more predictable network behavior. In order to develop intelligent routers for the next generation, a flexible and open software development environment is most important. We believe UNIX-based systems, once again, will play a vital role

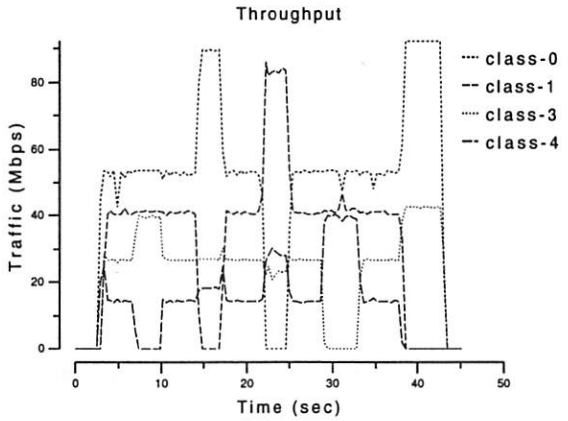


Figure 14: Link-Sharing Trace

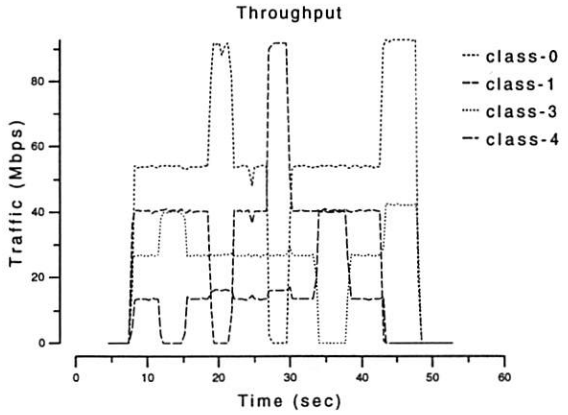


Figure 15: Link-Sharing Trace with Same Priority

in the advancement of technologies.

However, PC-UNIX based routers are unlikely to replace all router products in the market. Non-technical users will not use PC-UNIX based routers. High-speed routers or highly-reliable routers require special hardware and will not be replaced by PCs. Still, the advantage of PC-UNIX based routers is their flexibility and availability in source form, just like the advantage of UNIX over other operating systems. We argue that we should not let black boxes replace our routers and risk loosing our research and development environment. We should instead do our best to provide high-quality routers based on open technologies.

There are still many things that need to be worked out for the widespread use of PC-UNIX based routers. Network operators may worry about the reliability of PC-based routers. However, a reliable PC-based router can be built if the components are carefully selected. In most cases, problems are disk related troubles and it is possible to run UNIX without a hard disk by using a non-mechanical storage device such as ATA flash cards. Another reliability issue lies in PC components (e.g., cool-

ing fans) that may not be selected to be used 24 hours a day. There are a wide range of PC components and the requirements for a router are quite different from those for a desktop business machine. Some of the rack-mount PCs on the market are suitable for routers but SOHO routers need smaller chassis. We need PC hardware packages targeted for router use.

By the same token, we need software packages. It is not an easy task to configure a kernel to have the necessary modules for a router and make it run without a hard disk or a video card. Although there are many tools for routers, compilation, configuration, and maintenance of the tools are time consuming. Freely-available network administration tools seem to be weak but could be improved as PC-based routers become popular.

In summary, the technologies required to build quality PC-UNIX based routers are already available, but we need better packaging for both hardware and software. The networking research community would benefit a great deal if a line of PC-based router packages were available for specific scenarios, such as dial-up router, boundary router, workgroup router, etc.

6 Related Work

Our idea of providing a framework for queueing disciplines is not new. Nonetheless there have been few efforts to support a generic queueing framework. Research queueing implementations in the past have customized kernels for their disciplines [8, 19]. However, they are not generalized for use of other queueing disciplines.

ALTQ implements a switch to a set of queueing disciplines, which is similar to the protocol switch structure of BSD UNIX. A different approach is to use a modular protocol interface to implement a queueing discipline. STREAMS [17] and x-kernel [16] are such frameworks and the CBQ release for Solaris is actually implemented as a STREAMS module. Although it is technically possible to implement a queueing discipline as a protocol module, a queueing discipline is not a protocol and the requirements are quite different. One of the contributions of this paper is to have identified the requirements of a generic queueing framework.

A large amount of literature exists in the area of process scheduling but we are not concerned with process scheduling issues. Routers, as opposed to end hosts which run real-time applications, do not need real-time process scheduling because packet forwarding is part of interrupt processing. For end hosts, process scheduling is complementary to packet scheduling.

7 Current Status

The ALTQ implementation has been publicly available since March 1997. The current version runs on FreeBSD-2.2.x and implements CBQ, RED (including ECN), WFQ, and FIFOQ. The CBQ glues for ISI's RSVP are also included in the release. Most of the popular drivers including seven Ethernet drivers, one ATM driver, and three synchronous serial drivers can be used with ALTQ. ALTQ, as well as the original CBQ and RSVP, is still under active development.

ALTQ is used by many people as a research platform or a testbed. Although we do not know how many ALTQ users there are, our ftp server has recorded more than 1,000 downloads over the last 6 months. The majority of the users seem to use ALTQ for RSVP, but others do use ALTQ to control live traffic for their congested links and this group seems to be growing.

The performance of our implementation is quite satisfactory, but there are still many things to be worked out such as scalability issues in implementation and easier configuration for users.

We are planning to add new features, including support for IPv6, better support for slow links, better use of ATM VCs for traffic classes, and diskless configurations for more reliable router operations. Also, building good tools, especially traffic generators, is very important for development.

7.1 Availability

A public release of ALTQ for FreeBSD, the source code along with additional information, can be found at http://www.csl.sony.co.jp/person/kjc/software.html.

8 Conclusion

We have identified the requirements of a generic queueing framework and the issues of implementation. Then, we have demonstrated, with several queueing disciplines, that simple extension to BSD UNIX and minor fixes to drivers are enough to incorporate a variety of queueing disciplines. The main contribution of ALTQ is engineering efforts to make better queueing available for researchers and network operators on commodity PC platforms and UNIX. Our performance measurements clearly show the feasibility of traffic management by PC-UNIX based routers.

As router products have been proliferating over the last decade, network researchers have been losing research testbeds available in source form. We argue that general purpose computers, especially PC-based UNIX systems, have become once again competitive router platforms because of flexibility and cost/performance. Traffic management issues require technical innovations, and the key to progress is platforms which new ideas could be easily adopted into. ALTQ is an important step in that direction. We hope our implementation will stimulate other research activities in the field.

Acknowledgments

We would like to thank Elizabeth Zwicky and the anonymous reviewers for their helpful comments and suggestions on earlier drafts of this paper. We are also grateful to Sally Floyd for providing various information about CBQ and RED. We thank the members of Sony Computer Science Laboratory and the WIDE Project for their help in testing and debugging ALTQ. Hiroshi Kyusojin of Keio University implemented WFQ. The ALTQ release is a collection of outputs from other projects. These include CBQ and RED at LBNL, RSVP/CBQ at Sun, RSVP at ISI, and FreeBSD.

References

[1] Mary L. Baily, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkan. Pathfinder: A pattern-based packet classifier. In *Proceedings of Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, November 1994.

[2] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, K. L. Peterson, S. Shenker Ramakrishnan, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, IETF, April 1998.

[3] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM '89 Symposium*, pages 1–12, Austin, TX, September 1989.

[4] Sally Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5), October 1994.

[5] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–413, August 1993.

[6] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995. Also available from http://www-nrg.ee.lbl.gov/floyd/papers.html.

[7] The FreeBSD Project. http://www.freebsd.org/.

[8] Amit Gupta and Domenico Ferrari. Resource partitioning for real-time communication. *IEEE/ACM Transactions on Networking*, 3(5), October 1995. Also available from http://tenet.berkeley.edu/tenet-papers.html.

[9] The RSVP Project at ISI. http://www.isi.edu/rsvp/.

[10] Rick Jones. *Netperf: A Benchmark for Measuring Network Performance*. Hewlett-Packard Company, 1993. Available at http://www.cup.hp.com/netperf/NetperfPage.html.

[11] Srinivasan Keshav. On the efficient implementation of fair queueing. *Internetworking: Research and Experience*, 2:157–173, September 1991.

[12] P. E. McKenney. Stochastic fairness queueing. In *Proceedings of INFOCOM*, San Francisco, CA, June 1990.

[13] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Co., 1996.

[14] John Nagle. On packet switches with infinite storage. *IEEE Trans. on Comm.*, 35(4), April 1987.

[15] Abhay Parekh. A generalized processor sharing approach to flow control in integrated services networks. LIDS-TH 2089, MIT, February 1992.

[16] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. The x-kernel: A platform for accessing internet resources. *Computer*, 23(5):23–34, May 1990.

[17] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[18] McCanne S. and Floyd S. NS (Network Simulator). http://www-nrg.ee.lbl.gov/ns/, 1995.

[19] Ion Stoica and Hui Zhang. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proceedings of SIGCOMM '97 Symposium*, pages 249–262, Cannes, France, September 1997.

[20] Solaris RSVP/CBQ. ftp://playground.sun.com/pub/rsvp/.

[21] Ian Wakeman, Atanu Ghosh, Jon Crowcroft, Van Jacobson, and Sally Floyd. Implementing real-time packet forwarding policies using streams. In *Proceedings of USENIX '95*, pages 71–82, New Orleans, LA, January 1995.

[22] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7:8–18, September 1993. Also available from http://www.isi.edu/rsvp/pub.html.

Implementing Multiple Protection Domains in Java

Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski,
Deyu Hu, and Thorsten von Eicken

Department of Computer Science
Cornell University

Abstract

Safe language technology can be used for protection within a single address space. This protection is enforced by the language's type system, which ensures that references to objects cannot be forged. A safe language alone, however, lacks many features taken for granted in more traditional operating systems, such as rights revocation, thread protection, resource management, and support for domain termination. This paper describes the J-Kernel, a portable Java-based protection system that addresses these issues. J-Kernel protection domains can communicate through revocable capabilities, but are prevented from directly sharing unrevocable objects references. A number of micro-benchmarks are presented to characterize the costs of language-based protection, and an extensible web server based on the J-Kernel demonstrates the use of safe language techniques in a large application.

1 Introduction

Traditional operating systems use virtual memory to enforce protection between processes. A process cannot directly read and write other processes' memory, and communication between processes requires traps to the kernel. In the past decade of operating systems research, a large number of fast inter-process communication mechanisms have been proposed [3,8,25]. Nevertheless, the cost of passing through the kernel and of switching address spaces remains orders of magnitude larger than that of calling a procedure.

With the increasing adoption of extensible applications and component software, the cost of inter-process communication is leading to a difficult trade-off between robustness and performance. For example, the Netscape browser allows plug-ins to be loaded directly into the browser process to extend its functionality. However, an error in a plug-in can corrupt the entire browser. Although a separate process could be used for each plug-in, this would be both cumbersome to program and slow, because of the amount of communication between the plug-ins and the browser. Most web servers support plug-ins as well and in this case, the robustness issue is even more important—a browser crash may be annoying, but a server crash can be disastrous.

The robustness versus performance tradeoff is pervasive in component software (e.g., OLE, JavaBeans [16], ActiveX, OpenDoc). Microsoft's COM [27], for example, provides two different models for composing components: each component can run in its own process for protection, or multiple components can share a process (often termed *in-proc*) for performance. With more and more applications on the desktop being composed of "reusable" components the protection issue is becoming pressing: if not properly isolated, the failure of any component could cause large portions of the user's desktop environment to crash.

This paper explores the use of safe language technology to offer high performance as well as protection in a software component environment. Safe languages such as Java [12], Modula-3 [32], and CAML [22] use type safety and controlled linking to enforce protection between multiple components without relying on hardware support. In a safe language environment, calls across protection boundaries could potentially be as cheap as simple function calls, enabling as much communication between components as desired without performance drawbacks.

While many extensible applications and component environments can benefit from protection, the features required in different settings vary. In this paper, we assume that applications are composed of independently developed software components that communicate through well-structured interfaces. We assume that the protection mechanism should enforce this structure, just like modern languages enforce module or class structures. Thus communication should only be possible through well-defined interfaces, and not through side effects. In all settings, we strive to enable failure isolation: a bug in one component should not crash other components. However, the required degree of failure isolation varies: in an application suite produced by a group of developers, the primary concern is accidental effects of one component on another. On the other hand, a web server allowing arbitrary users to upload extensions requires bulletproof protection to guard against malicious behavior.

Several projects [1,7,9,13,41] have recently described how to build protection domains around components in a safe language environment, where a protection domain specifies the resources to which a software component has access. The central ideas are to use the linker to create multiple namespaces and to use object references (i.e., pointers to objects) as capabilities for cross-domain communication. The multiple namespaces ensure that the same variable, procedure, or type names can refer to different instances in different domains. Object references in safe languages are unforgeable and can thus be used to confer certain rights to the holder(s). In an object-oriented language, the methods applicable to an object are in essence call gates. This paper argues in Section 2 that this straightforward approach, while both flexible and fast, is ultimately unsatisfactory: using objects references as capabilities leads to severe problems with revocation, resource management, and inter-domain dependency analysis.

In order to overcome the limitations of the straightforward approach, we introduce additional mechanisms borrowed from traditional capability systems. The result is a system described in Section 3, called the J-Kernel. The J-Kernel is written entirely in Java, and provides sophisticated capability-based protection features. We choose Java for practical reasons—Java is emerging as the most widely used general-purpose safe language, and dependable Java virtual machines are widespread and easy to work with. While Java does allow for multiple protection domains within a single Java Virtual Machine (JVM) using the sandbox model for applets, that model is currently very restrictive. It lacks many of the characteristics that are taken for granted in more traditional systems and, in particular, does not provide a clear way for different protection domains to communicate with each other.

We concentrated our efforts on developing a general framework to allow multiple protection domains within a single JVM. We provide features found in traditional operating systems, such as support for rights revocation and domain termination. In addition, we support flexible protection policies between components, including support for communication between mutually suspicious components. The main benefits of our system are a highly flexible protection model, low overheads for communication between software components, and operating system independence. Our current J-Kernel implementation runs on standard JVMs.

Language-based protection does have drawbacks. First, code written in a safe language tends to run more slowly than code written in C or assembly language, and thus the improvement in cross-domain communication may be offset by an overall slowdown. While much of this slowdown is due to current Java just-in-time compilers optimizing for fast compile times at the expense of run-time performance, even with sophisticated optimization it seems likely that Java programs will not run as fast as C programs. Second, all current language-based protection systems are designed around a single language, which limits developers and doesn't handle legacy code. Software fault isolation [40] and verification of assembly language [29,30,31] may someday offer solutions, but are still an active area of research [37].

Section 4 describes an extensible web server based on the J-Kernel. Section 5 discusses related work, and section 6 concludes.

2 Language-based protection background

In an unsafe language, any code running in an address space can potentially modify any memory location in that address space. While, in theory, it is possible to prove that certain pieces of code only modify a restricted set of memory locations, in practice this is very difficult for languages like C and arbitrary assembly language [4, 30], and cannot be fully automated. In contrast, the type system and the linker in a safe language restrict what operations a particular piece of code is allowed to perform on which memory locations.

The term *namespace* can be used to express this restriction: a namespace is a partial function mapping names of operations to the actions taken when the operations are executed. For example, the operation "read the field `out` from the class `System`" may perform different actions depending on what class the name `System` refers to.

Protection domains around software components can be constructed in a safe language system by providing a separate namespace for each component. Communication between components can then be enabled by introducing sharing among namespaces. Java provides three basic mechanisms for controlling namespaces: selective sharing of object references, static access controls, and selective class sharing.

Selective sharing of object references

Two domains can selectively share references to objects by simply passing each other these references. In the example below, `method1` of class A creates two objects of type A, and passes a reference to the first object to `method2` of class B. Since `method2` acquires a reference to `a1`, it can perform operations on

it, such as incrementing the field `j`. However, `method2` was not given a reference to `a2` and thus has no way of performing any operations on it. Java's safety prevents `method2` from forging a reference to `a2`, e.g., by casting an integer holding `a2`'s address to a pointer.

```
class A {
    private int i;
    public int j;
    public static void method1() {
        A a1 = new A();
        A a2 = new A();
        B.method2(a1);
    }
}

class B {
    public static void method2(A arg) {
        arg.j++;
    }
}
```

Static access control

The preceding example demonstrated a very dynamic form of protection—methods can only perform operations on objects to which they have been given a reference. Java also provides static protection mechanisms that limit what operations a method can perform on an object once the method has acquired a reference to that object. A small set of modifiers can change the scope of fields and methods of an object. The two most common modifiers, `private` and `public`, respectively limit access to methods in the same class or allow access to methods in any class. In the classes shown above, `method2` can access the `public` field `j` of the object `a1`, but not the `private` field `i`.

Selective class sharing

Domains can also protect themselves through control of their *class namespace*. To understand this, we need to look at Java's class loading mechanisms. To allow dynamic code loading, Java supports user-defined *class loaders* which load new classes into the virtual machine at run-time. A class loader fetches Java bytecode from some location, such as a file system or a URL, and submits the bytecode to the virtual machine. The virtual machine performs a verification check to make sure that the bytecode is legal, and then integrates the new class into the machine execution. If the bytecode contains references to other classes, the class loader is invoked recursively in order to load those classes as well.

Class loaders can enforce protection by making some classes visible to a domain, while hiding others. For instance, the example above assumed that classes A and B were visible to each other. However, if class A were hidden from class B (i.e. it did not appear in B's class namespace), then even if B obtains a reference to an object of type A, it will not be able to access the fields `i` and `j`, despite the fact that `j` is public.

2.1 Straight-forward protection domains: the *share anything* approach

The simple controls over the namespace provided in Java can be used to construct software components that communicate with each other but are still protected from one another. In essence, each component is launched in its own namespace, and can then share any class and any object with other components using the mechanisms described above. While we will continue to use the term *protection domain* informally to refer to these protected components, we will argue that it is impossible to precisely define protection domains when using this approach.

The example below shows a hypothetical file system component that gives objects of type `FileSystemInterface` to its clients to give them access to files. Client domains make cross-domain invocations on the file system by invoking the `open` method of a `FileSystemInterface` object. By specifying different values for `accessRights` and `rootDirectory` in different objects, the file system can enforce different protection policies for different clients. Static access control ensures that clients cannot modify the `accessRights` and `rootDirectory` fields directly, and one client cannot forge a reference to another client's `FileSystemInterface` object.

```
class FileSystemInterface {
    private int accessRights;
    private Directory rootDirectory;
    public File open(String fileName) {…}
}
```

The filesystem example illustrates an approach to protection in Java that resembles a capability system. Several things should be noted about this approach. First, this approach does not require any extensions to the Java language—all the necessary mechanisms already exist. Second, there is very little overhead involved in making a call from one protection domain to another, since a cross-domain call is simply a method invocation, and large arguments can be passed by reference, rather than by copy. Third, references to any object may be shared between domains since the Java language has no way of restricting which references can be passed through a cross-domain method invocation and which cannot.

When we first began to explore protection in Java, this *share anything* approach seemed the natural basis for a

protection system, and we began developing on this foundation. However, as we worked with this approach a number of problems became apparent.

Revocation

The first problem is that access to an object reference cannot be revoked. Once a domain has a reference to an object, it can hold on to it forever. Revocation is important in enforcing the principle of least privilege: without revocation, a domain can hold onto a resource for much longer than it actually needs it.

The most straightforward implementation of revocation uses extra indirection. The example below shows how a revocable version of the earlier class `A` can be created. Each object of `A` is wrapped with an object of `AWrapper`, which permits access to the wrapped object only until the `revoked` flag is set.

```
class A {
  public int meth1(int a1, int a2) {…}
}

class AWrapper {
  private A a;
  private boolean revoked;
  public int meth1(int a1, int a2) {
    if(!revoked) return a.meth1(a1, a2);
    else throw new RevokedException();
  }
  public void revoke() { revoked=true; }
  public AWrapper(A realA) {
    a = realA; revoked = false; }
}
```

In principle, this solves the revocation problem and is efficient enough for most purposes. However, our experience shows that programmers often forget to wrap an object when passing it to another domain. In particular, while it is easy to remember to wrap objects passed as arguments, it is common to forget to wrap other objects to which the first one points. In effect, the default programming model ends up being an unsafe model where objects cannot be revoked. This is the opposite of the desired model: safe by default and unsafe only in special cases.

Inter-domain Dependencies and Side Effects

As more and more object references are shared between domains, the structure of the protection domains is blurred, because it is unclear from which domains a shared object can be accessed. For the programmer, it becomes difficult to track which objects are shared between protection domains and which are not, and the Java language provides no help as it makes no distinction between the two. Yet, the distinction is critical for reasoning about the behavior of a program running in a domain. Mutable shared objects can be modified at any time in by other domains that have access to the object, and a programmer needs to be aware of this possible activity. For example, a malicious user might try to pass a byte array holding legal bytecode to a class loader (byte arrays, like other objects, are passed by reference to method invocations), wait for the class loader to verify that the bytecode is legal, and then overwrite the legal bytecode with illegal bytecode which would subsequently be executed. The only way the class loader can protect itself from such an attack is to make its own private copy of the bytecode, which is not shared with the user and is therefore safe from malicious modification.

Domain Termination

The problems associated with shared object references come to a head when we consider what happens when a domain must be terminated. Should all the objects that the domain allocated be released, so that the domain's memory is freed up? Or should objects allocated by the domain be kept alive as long as other domains still hold references to them? From a traditional operating systems perspective, it seems natural that when a process terminates all of its objects disappear, because the address space holding those objects ceases to exist. On the other hand, from a Java perspective, objects can only be deallocated when there are no more reachable references to them.

Either solution to domain termination leads to problems. Deallocating objects when the domain terminates can be extremely disruptive if objects are shared at a fine-grained level and there is no explicit distinction between shared and non-shared objects. For example, consider a Java `String` object, which holds an internal reference to a character array object. Suppose domain 2 holds a `String` object whose internal character array belongs to domain 1. If domain 1 dies, then the `String` will suddenly stop working, and it may be beyond the programmer's ability to deal with disruptions at this level.

On the other hand, if a domain's objects do not disappear when the domain terminates, other problems can arise. First, if a server domain fails, its clients may continue to hold on to the server's objects and attempt to continue using them. In effect, the server's failure is not propagated correctly to the clients. Second, if a client domain holds on to a server's objects, it may indirectly also hold on to other resources, such as open network connections and files. A careful server implementation could explicitly relinquish important resources before exiting, but in the case of unexpected termination this may be impossible. Third, if one domain holds on to another domain's objects after the latter exits, then any memory leaks in the terminated domain may be unintentionally transferred to the remaining one. It is easy to imagine scenarios where

recovery from this sort of shared memory leak requires a shutdown of the entire VM.

Threads

By simply using method invocation for cross-domain calls, the caller and callee both execute in the same thread, which creates several potential hazards. First, the caller must block until the callee returns – there is no way for the caller to gracefully back out of the call without disrupting the callee's execution. Second, Java threads support methods such as `stop`, `suspend`, and `setPriority` that modify the state of a thread. A malicious domain could call another domain and then suspend the thread so that the callee's execution gets blocked, perhaps while holding a critical lock or other resource. Conversely, a malicious callee could hold on to a `Thread` object and modify the state of the thread after execution returns to the caller.

Resource Accounting

A final problem with the simple protection domains is that object sharing makes it difficult to hold domains accountable for the resources that they use, such as processor time and memory. In particular, it is not clear how to define a domain's memory usage when domains share objects. One definition is that a domain is held accountable for all of the objects that it allocates, for as long as those objects remain alive. However, if shared objects aren't deallocated when the domain exits, a domain might continue to be charged for shared objects that it allocated, long after it has exited. Perhaps the cost of shared objects should be split between all the domains that have references to the object. However, because objects can contain references to other objects, a malicious domain could share an object that looks small, but actually contains pointers to other large objects, so that other domains end up being charged for most of the resources consumed by the malicious domain.

Summary

The simple approach to protection in Java outlined in this section is both fast and flexible, but it runs into trouble because of its lack of structure. In particular, it fails to clearly distinguish between the ordinary, non-shared object references that constitute a domain's internal state, and the shared object references that are used for cross-domain communication. Nevertheless, this approach is useful to examine, because it illustrates how much protection is possible with the mechanisms provided by the Java language itself. It suggests that the most natural approach to building a protection system in Java is to make good use of the language's inherent protection mechanisms, but to introduce additional structure to fix the problems. The next section presents a system that retains the flavor of the simple approach, but makes a stronger distinction between non-shared and shared objects.

3 The J-Kernel

The J-Kernel is a capability-based system that supports multiple, cooperating protection domains which run inside a single Java virtual machine. Capabilities were chosen because they have several advantages over access lists: (i) they can be implemented naturally in a safe language, (ii) they can enforce the principle of least privilege more easily, and (iii) by avoiding access list lookups, operations on capabilities can execute quickly.

The primary goals of the J-Kernel are:

- a precise definition of protection domains, with a clear distinction between objects local to a domain and *capability objects* that can be shared between domains,
- well defined, flexible communication channels between domains based on capabilities,
- support for revocation for all capabilities, and
- clean semantics of domain termination.

To achieve these goals, we were willing to accept higher cross-domain communication overheads when compared to the share anything approach. In order to ensure portability, the J-Kernel is implemented entirely as a Java library and requires no native code or modifications to the virtual machine. To accomplish this, the J-Kernel defines a class loader that examines and in some cases modifies user-submitted bytecode before passing it on to the virtual machine. This class loader also generates bytecode at run-time for stub classes used for cross-domain communication. Finally, the J-Kernel's class loader substitutes safe versions for some problematic standard classes. With these implementation techniques, the J-Kernel builds a protection architecture that is radically different from the security manager based protection architecture that is the default model on most Java virtual machines.

Protection in the J-Kernel is based on three core concepts—capabilities, protection domains, and cross-domain calls:

- *Capabilities* are implemented as objects of the class `Capability` and represent handles onto resources in other domains. A capability can be revoked at any time by the domain that created it. All uses of a revoked capability throw an exception, ensuring the correct propagation of failure.
- *Protection domains* are represented by the Java class `Domain`. Each protection domain has a namespace that it controls as well as a set of threads. When a domain terminates, all of the capabilities that it

created are revoked, so that all of its memory may be freed, thus avoiding the domain termination problems that plagued the share anything approach.

- *Cross-domain calls* are performed by invoking methods of capabilities obtained from other domains. The J-Kernel's class loader interposes a special calling convention[1] for these calls: arguments and return values are passed by reference if they are also capabilities, but they are passed by copy if they are primitive types or non-capability objects. When an object is copied, these rules are applied recursively to the data in the object's fields, so that a deep copy of the object is made. The effect is that only capabilities can be shared between protection domains and references to regular objects are confined to single domains.

3.1 J-Kernel implementation

The J-Kernel's implementation of capabilities and cross-domain calls relies heavily on Java's *interface classes*. An interface class defines a set of method signatures without providing their implementation. Other classes that provide corresponding implementations can then be declared to *implement* the interface. Normally interface classes are used to provide a limited form of multiple inheritance (properly called interface inheritance) in that a class can implement multiple interfaces. In addition, Sun's remote method invocation (RMI) specification [17] "pioneered" the use of interfaces as compiler annotations. Instead of using a separate interface definition language (IDL), the RMI specification simply uses interface classes that are flagged to the RMI system in that they extend the class `Remote`. Extending `Remote` has no effect other than directing the RMI system to generate appropriate stubs and marshalling code.

Because of the similarity of the J-Kernel's cross-domain calls to remote method invocations, we have integrated much of Sun's RMI specification into the capability interface. The example below shows a simple *remote interface* and a class that implements this remote interface, both written in accordance with Sun's RMI specification.

```
// interface class shared with other domains
interface ReadFile extends Remote {
  byte readByte() throws RemoteException;
  byte[] readBytes(int nBytes)
    throws RemoteException;
}
```

[1] The standard Java calling convention passes primitive data types (int, float, etc.) by copy and object data types by reference.

```
// implementation hidden from other domains
class ReadFileImpl implements ReadFile {
  public byte readByte() {…}
  public byte[] readBytes(int nBytes) {…}
  …
}
```

To create a capability in the J-Kernel, a domain calls the `create` method of the class `Capability`, passing as an argument a target object that implements one or more remote interfaces. The `create` method returns a new capability, which extends the class `Capability` and implements all of the remote interfaces that the target object implements. The capability can then be passed to other domains, which can cast it to one of its remote interfaces, and invoke the methods this interface declares. In the example below domain 1 creates a capability and adds it to the system-wide repository (the repository is a name service allowing domains to publish capabilities). Domain 2 retrieves the capability from the repository, and makes a cross-domain invocation on it.

Domain 1:

```
// instantiate new ReadFileImpl object
ReadFileImpl target = new ReadFileImpl();
// create a capability for the new object
Capability c = Capability.create(target);
// add it to repository under some name
Domain.getRepository().bind(
    "Domain1ReadFile", c);
```

Domain 2:

```
// extract capability
Capability c = Domain.getRepository()
  .lookup("Domain1ReadFile");
// cast it to ReadFile, and invoke remote method
byte b = ((ReadFile) c).readByte();
```

Essentially, a capability object is a wrapper object around the original target object. The code for each method in the wrapper switches to the domain that created the capability, makes copies of all non-capability arguments according to the special calling convention, and then invokes the corresponding method in the target object. When the target object's method returns, the wrapper switches back to the caller domain, makes a copy of the return value if it is not a capability, and returns.

Local-RMI stubs

The simple looking call to `Capability.create` in fact hides most of the complexity of traditional RPC systems. Internally, `create` automatically generates a stub class at run-time for each target class. This avoids off-line stub generators and IDL files, and it allows the J-Kernel to specialize the stubs to invoke the target methods with minimal overhead. Besides switching domains, stubs have three roles: copying arguments, supporting revocation, and protecting threads.

By default, the J-Kernel uses Java's built-in serialization features [17] to copy an argument: the J-Kernel serializes an argument into an array of bytes, and then deserializes the byte array to produce a fresh copy of the argument. While this is convenient because many built-in Java classes are serializable, it involves a substantial overhead. Therefore, the J-Kernel also provides a fast copy mechanism, which makes direct copies of objects and their fields without using an intermediate byte array. The fast copy implementation automatically generates specialized copy code for each class that the user declares to be a fast copy class. For cyclic or directed graph data structures, a user can request that the fast copy code use a hash table to track object copying, so that objects in the data structure are not copied more than once (this slows down copying, though, so by default the copy code does not use a hash table).

Each generated stub contains a revoke method that sets the internal pointer to the target object to `null`. Thus all capabilities can be revoked and doing so makes the target object eligible for garbage collection, regardless of how many other domains hold a reference to the capability. This prevents domains from holding on to garbage in other domains.

In order to protect the caller's and callee's threads from each other, the generated stubs provide the illusion of switching threads. Because most virtual machines map Java threads directly onto kernel threads it is not practical to actually switch threads: as shown in the next subsection this would slow down cross-domain calls substantially. A fast user-level threads package might solve this problem, but would require modifications to the virtual machine, and would thus limit the J-Kernel's portability. The compromise struck in the current implementation uses a single Java thread for both the caller and callee but prevents direct access to that thread to avoid security problems.

Conceptually, the J-Kernel divides each Java thread into multiple segments, one for each side of a cross-domain call. The J-Kernel class loader then hides the system `Thread` class that manipulates Java threads, and interposes its own with an identical interface but an implementation that only acts on the local thread segment. Thread modification methods such as `stop` and `suspend` act on thread segments rather than Java threads, which prevents the caller from modifying the callee's thread segment and vice-versa. This provides the illusion of thread-switching cross-domain calls, without the overhead for actually switching threads. The illusion is not totally convincing, however – cross-domain calls really do block, so there is no way for the caller to gracefully back out of one if the callee doesn't return.

Class Name Resolvers

In the standard Java applet architecture, applets have very little access to Java's class loading facilities. In contrast, J-Kernel domains are given considerable control over their own class loading. Each domain has its own class namespace that maps names to classes. Classes may be local to a domain, in which case they are only visible in that domain's namespace or they may be shared between multiple domains, in which case they are visible in many namespaces. A domain's namespace is controlled by a user-defined *resolver*, which is queried by the J-Kernel whenever a new class name is encountered. A domain can use a resolver to load new bytecode into the system, or it can make use of existing shared classes. After a domain has loaded new classes into the system, it can share these classes with other domains if it wants, by making a `SharedClass` capability available to other domains[2].

Shared classes are the basis for cross-domain communication: domains must share remote interfaces and fast copy classes to establish common methods and argument types for cross-domain calls. Allowing user-defined shared classes makes the cross-domain communication architecture extensible; standard Java security architectures only allow pre-defined "system classes" to be shared between domains, and thus limit the expressiveness of cross-domain communication.

Ironically, the J-Kernel needs to *prevent* the sharing of some system classes. For example, the file system and thread classes present security problems. Others contain resources that need to be defined on a per-domain basis: the class `System`, for example, contains static fields holding the standard input/output streams. In other words, the "one size fits all" approach to class sharing in most Java security models is simply not adequate, and a more flexible model is essential to make the J-Kernel safe, extensible, and fast.

In general, the J-Kernel tries to minimize the number of system classes visible to domains. Classes that would normally be loaded as system classes (such as classes containing native code) are usually loaded into a privileged domain in the J-Kernel, and are accessed through cross-domain communication, rather than through direct calls to system classes. For instance, we have developed a domain for file system access that is called using cross-domain communication. To keep

[2] Shared classes (and, transitively, the classes that shared classes refer to) are not allowed to have static fields, to prevent sharing of non-capability objects through static fields. In addition, to ensure consistency between domains, two domains that share a class must also share other classes referenced by that class.

compatibility with the standard Java file API, we have also written alternate versions of Java's standard file classes, which are just stubs that make the necessary cross-domain calls. (This is similar to the interposition proposed by [41]).

The J-Kernel moves functionality out of the system classes and into domains for the same reasons that micro-kernels move functionality out of the operating system kernel. It makes the system as a whole extensible, i.e., it is easy for any domain to provide alternate implementations of most classes that would normally be system classes (such as file, network, and thread classes). It also means that each such service can implement its own security policy. In general, it leads to a cleaner overall system structure, by enforcing a clear separation between different modules. Java libraries installed as system classes often have undocumented and unpredictable dependencies on one another[3]. Richard Rashid warned that the UNIX kernel had "become a 'dumping ground' for every new feature or facility"[34]; it seems that the Java system classes are becoming a similar dumping ground.

3.2 J-Kernel Micro-Benchmarks

To evaluate the performance of the J-Kernel mechanisms we measured a number of micro-benchmarks on the J-Kernel as well as on a number of reference systems. Unless otherwise indicated, all micro-benchmarks were run on 200Mhz Pentium-Pro systems running Windows NT 4.0 and the Java virtual machines used were Microsoft's VM (MS-VM) and Sun's VM with Symantec's JIT compiler (Sun-VM). All numbers are averaged over a large number of iterations.

Null LRMI

Table 1 dissects the cost of a null cross-domain call (null LRMI) and compares it to the cost of a regular method invocation, which takes a few tens of nanoseconds. The J-Kernel null LRMI takes 60x to 180x longer than a regular method invocation. With MS-VM, a significant fraction of the cost lies in the interface method invocation necessary to enter the stub. Additional overheads include the synchronization cost when changing thread segments (two lock acquire/release pairs per call) and the overhead of looking up the current thread. Overall, these three operations account for about 70% of the cross-domain call on MS-VM and about 80% on Sun-VM. Given that the implementations of the three operations are independent, we expect significantly better performance in a system that includes the best of both VMs.

Operation	MS-VM	Sun-VM
Regular Method invocation	0.04μs	0.03μs
Interface method invocation	0.54μs	0.05μs
Thread info lookup	0.55μs	0.29μs
Acquire/release lock	0.20μs	1.91μs
J-Kernel LRMI	2.22μs	5.41μs

Table 1. Cost of null method invocations

To compare the J-Kernel LRMI with traditional OS cross-domain calls, Table 2 shows the cost of several forms of local RPC available on NT. *NT-RPC* is the standard, user-level RPC facility. *COM out-of-proc* is the cost of a null interface invocation to a COM component located in a separate process on the same machine. The communication between two fully protected components is at least a factor of 3000 from a regular C++ invocation (shown as *COM in-proc*).

Form of RPC	Time
NT-RPC	109μs
COM out-of-proc	99μs
COM in-proc	0.03μs

Table 2. Local RPC costs using NT mechanisms

Threads

Table 3 shows the cost of switching back and forth between two Java threads in MS-VM and Sun-VM. The base cost of two context switches between NT kernel threads (*NT-base*) is 8.6μs, and Java introduces an additional 1-2μs of overhead. This confirms that switching Java threads during cross-domain calls would add a significant cost to J-Kernel LRMI.

NT-base	MS-VM	Sun-VM
8.6μs	9.8μs	10.2μs

Table 3. Cost of a double thread switch using regular Java threads

Argument Copying

Table 4 compares the cost of copying arguments during a J-Kernel LRMI using Java serialization and using the J-Kernel's fast-copy mechanism. By making direct copies of the objects and their fields without using an intermediate Java byte-array, the fast-copy mechanism improves the performance of LRMI substantially—more than an order of magnitude for large arguments. The performance difference between the second and third rows (both copy the same number of bytes) is due

[3] For instance, Microsoft's implementation of `java.io.File` depends on `java.io.DataInputStream`, which depends on `com.ms.lang.SystemX`, which depends on classes in the abstract windowing toolkit. Similarly, `java.lang.Object` depends transitively on almost every standard library class in the system.

to the cost of object allocation and invocations of the copying routine for every object.

Number of objects and size	MS-VM		Sun-VM	
	LRMI w/ Serial.	LRMI w/ Fast-copy	LRMI w/ Serial	LRMI w/ Fast-Copy
1 x 10 bytes	104μs	4.8μs	331μs	13.7μs
1 x 100 bytes	158μs	7.7μs	509μs	18.5μs
10 x 10 bytes	193μs	23.3μs	521μs	79.3μs
1 x 1000 bytes	633μs	19.2μs	2105μs	66.7μs

Table 4. Cost of Argument Copying

In summary, the micro-benchmark results are encouraging in that the cost of a cross-domain call is 50x lower in the J-Kernel than in NT. However, the J-Kernel cross-domain call still incurs a stiff penalty over a plain method invocation due to the lack of optimizations in Java.

4 An Extensible Http Server

One of the driving applications for the J-Kernel is an extensible HTTP server. The goal is to allow users to dynamically extend the functionality of the server by uploading Java programs, called *servlets* [19], that customize the HTTP request processing for a subset of the server's URL space.

Instead of building (or porting) an entire HTTP server in Java, we integrated the J-Kernel into the off-the-shelf Microsoft server (IIS 3.0). The J-Kernel runs within the same process as IIS (as an in-proc ISAPI extension) and includes a system servlet with access to native methods that allows it to receive HTTP requests from IIS and return corresponding replies. This HTTP system servlet forwards each request to the appropriate user servlet, each of which runs in its own J-Kernel domain. The implementation of the bridge between IIS and the J-Kernel is multithreaded to allow multiple outstanding HTTP requests and it allows the Java code to run in the same thread as IIS uses to invoke the bridge.

Server throughput measurements

To quantify the impact of the J-Kernel overheads in the performance of the HTTP server, several simple experiments measure the number of documents per second that can be served by Microsoft's IIS, Sun's Java Web Server 1.0.2 (JWS) [20], and J-Kernel running inside IIS. The hardware platform consists of a quad-processor 200MHz Pentium-Pro (results obtained on one- and two-processor machines are similar). The parameter of the experiments is the size of document being served. All three tests follow the same scenario: eight multithreaded clients repeatedly request the same document. IIS serves documents in a traditional way—by fetching them from NT's file cache, while JWS and J-Kernel utilize servlets to return in-memory documents.

Table 5 shows that the overhead of passing requests into and out of the J-Kernel decreases IIS's performance by 20%. Additional measurements show that the ISAPI bridge accounts for about half of that performance gap and only the remainder is directly attributable to the J-Kernel. The order-of-magnitude gap between J-Kernel and JWS is due to the fact that JWS is written entirely in Java and is executed without a JIT compiler. At the time of this writing the implementation of JWS as an IIS plug-in with JIT was not fully functional.

Page size	IIS	JWS	IIS+J-Kernel
10 bytes	801	122	662
100 bytes	790	121	640
1000 bytes	759	96	616

Table 5. HTTP server throughput in pages/second

5 Related Work

Several major vendors have proposed extensions to the basic Java sandbox security model for applets [18, 33, 28]. For instance, Sun's JDK 1.1 added a notion of authentication, based on code signing, while the JDK 1.2 adds a richer structure for authorization, including classes that represent permissions and methods that perform access control checks based on stack introspection [11]. JDK 1.2 "protection domains" are implicitly created based on the origin of the code, and on its signature. This definition of a protection domain is closer to a *user* in Unix, while the J-Kernel's protection domain is more like a *process* in Unix. Balfanz et al. [1] define an extension to the JDK which associates domains with users running particular code, so that a domain becomes more like a process. However, if domains are able to share objects directly, revocation, resource management, and domain termination still need to be addressed in the JDK.

JDK 1.2 system classes are still lumped into a monolithic "system domain", but a new classpath facilitates loading local applications with class loaders rather than as system classes. However, only system classes may be shared between domains that have different class loaders, which limits the expressiveness of communication between domains. In contrast, the J-Kernel allows domains to share classes without requiring these domains to use the same class loader. In the future work section, Gong et al. [11] mentions separating current system classes (such as file classes) into separate domains, in accordance with the principle of least privilege. The J-Kernel already moves facilities

for files and networking out of the system classes and into separate domains.

A number of related safe-language systems are based on the idea of using object references as capabilities. Wallach et. al. [41] describe three models of Java security: type hiding (making use of dynamic class loading to control a domain's namespace), stack introspection, and capabilities. They recommended a mix of these three techniques. The E language from Electric Communities [7] is an extension of Java targeted towards distributed systems. E's security architecture is capability based; programmers are encouraged to use object references as the fundamental building block for protection. Odyssey [9] is a system that supports mobile agents written in Java; agents may share Java objects directly. Hagimont et al. [13] describe a system to support capabilities defined with special IDL files. All three of these systems allow non-capability objects to be passed directly between domains, and generally correspond to the share anything approach described in Section 2. They do not address the issues of revocation, domain termination, thread protection, or resource accounting.

The SPIN project [2] allows safe Modula-3 code to be downloaded into the operating system kernel to extend the kernel's functionality. SPIN has a particularly nice model of dynamic linking [39] to control the namespace of different extensions. Since it uses Modula-3 pointers directly as capabilities, the limitations of the share anything approach apply to it.

Several recent software-based protection techniques do not rely on a particular high level language like Java or Modula-3. Typed assembly language [29] pushes type safety down to the assembly language level, so that code written at the assembly language level can be statically type checked and verified as safe. Software fault isolation [40] inserts run-time "sandboxing" checks into binary executables to restrict the range of memory that is accessible to the code. With suitable optimizations, sandboxed code can run nearly as fast as the original binary on RISC architectures. However, it is not clear how to extend optimized sandboxing techniques to CISC architectures, and sandboxing cannot enforce protection at as fine a granularity as a type system. Proof carrying code [30, 31] generalizes many different approaches to software protection—arbitrary binary code can be executed as long as it comes with a proof that it is safe. While this can potentially lead to safety without overhead, generating the proofs for a language as complex as Java is still a research topic.

The J-Kernel enforces a structure that is similar to traditional capability systems [21,23]. Both the J-Kernel and traditional capability systems are founded on the notion of unforgeable capabilities. In both, capabilities name objects in a context-independent manner, so that capabilities can be passed from one domain to another. The main difference is that traditional capability systems used virtual memory or specialized hardware support to implement capabilities, while the J-Kernel uses language safety. The use of virtual memory or specialized hardware led either to slow cross-domain calls, to high hardware costs, or to portability limitations. Using Java as the basis for the J-Kernel simplifies many of the issues that plagued traditional capability systems. First, unlike systems based on capability lists, the J-Kernel can store capabilities in data structures, because capabilities are implemented as Java objects. Second, rights amplification [21] is implicit in the object-oriented nature of Java: invocations are made on methods, rather than functions, and methods automatically acquire rights to their *self* parameter. In addition, selective class sharing can be used to amplify other parameters. Although many capability systems did not support revocation, the idea of using indirection to implement revocation goes back to Redell [35]. The problems with resource accounting were also on the minds of implementers of capability systems—Wulf et. al. [42] point out that "No one 'owns' an object in the Hydra scheme of things; thus it's very hard to know to whom the cost of maintaining it should be charged".

Single-address operating systems, like Opal [5] and Mungi [15], remove the address space borders, allowing for cheaper and easy sharing of data between processes. Opal and Mungi were implemented on architectures offering large address spaces (64-bit) and used password capabilities as the protection mechanism. Password capabilities are protected from forgery by a combination of encryption and sparsity.

Several research operating systems support very fast inter-process communication. Recent projects, like L4, Exokernel, and Eros, provide fine-tuned implementations of selected IPC mechanisms, yielding an order of magnitude improvement over traditional operating systems. The systems are carefully tuned and aggressively exploit features of the underlying hardware.

The L4 μ–kernel [14] rigorously aims for minimality and is designed from scratch, unlike first-generation μ–kernels, which evolved from monolithic OS kernels. The system was successful at dispelling some common misconceptions about μ–kernel performance limitations. Exokernel [8] shares L4's goal of being an ultra-fast "minimalist" kernel, but is also concerned with untrusted loadable modules (similar to the SPIN project). Untrusted code is given efficient control over hardware resources by separating management from

protection. The focus of the EROS [38] project is to support orthogonal persistence and real-time computations. Despite quite different objectives, all three systems manage to provide very fast implementations of IPC with comparable performance, as shown in Table 6. A short explanation of the 'operation' column is needed. Round-trip IPC is the time taken for a call transferring one byte from one process to another and returning to the caller; Exokernel's protected control transfer installs the callee's processor context and starts execution at a specified location in the callee.

The results are contrasted with a 3-argument method invocation in the J-Kernel. The J-Kernel's performance is comparable with the three very fast systems. It is important to note that L4, Exokernel and Eros are implemented as a mix of C and assembly language code, while J-Kernel consists of Java classes without native code support. Improved implementations of JVMs and JITs are likely to enhance the performance of the J-Kernel.

System	Operation	Platform	μs
L4	Round-trip IPC	P5-133	1.82
Exokernel	Protected control transfer (r/t)	DEC-5000	2.40
Eros	Round-trip IPC	P5-120	4.90
J-Kernel	Method invocation with 3 args	P5-133	3.77

Table 6. Comparison with selected kernels.

6 Conclusion

This paper explores the use of safe language technology to construct robust protection domains. The advantages of using language-enforced protection are portability and good cross-domain performance. The most straightforward implementation of protection in a safe language environment is to use object references directly as capabilities. However, problems of revocation, domain termination, thread protection, and resource accounting arise when non-shared object references are not clearly distinguished from shared capabilities. We argue that a more structured approach is needed to solve these problems: only capabilities can be shared, and non-capability objects are confined to single domains.

We developed the J-Kernel system, which demonstrates how the issues of object sharing, class sharing, and thread protection can be addressed. As far as we know, the J-Kernel is the first Java-based system that integrates solutions to these issues into a single, coherent protection system. Our experience using the J-Kernel to extend the Microsoft IIS web server leads us to believe that a safe language system can achieve both robustness and high performance. Simple servlets downloaded into the web server achieve a performance close to that of the server running stand-alone.

Because of its portability and flexibility, language-based protection is a natural choice for a variety of extensible applications and component-based systems. From a performance point of view, safe language techniques are competitive with fast microkernel systems, but do not yet achieve their promise of making cross-domain calls as cheap as function calls. Implementing a stronger model of protection than the straightforward share anything approach leads to thread management costs and copying costs, which increase the overhead to much more than a function call. Fortunately, there clearly is room for improvement. We found that many small operations in Java, such as allocating an object, invoking an interface method, and manipulating a lock were slower than necessary on current virtual machines. Java just-in-time compiler technology is still evolving. We expect that as virtual machine performance improves, the J-Kernel's cross-domain performance will also improve. In the meantime, we will continue to explore optimizations possible on top of current off-the-shelf virtual machines, as well as to examine the performance benefits that customizing the virtual machine could bring.

Acknowledgments

The authors would like to thank Greg Morrisett, Fred Schneider, Fred Smith, Lidong Zhou, and the anonymous reviewers for their comments and suggestions. This research is funded by DARPA ITO contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations. Chi-Chao Chang is supported in part by a doctoral fellowship (200812/94-7) from CNPq/Brazil.

7 References

1. Balfanz, D. and Gong, L. *Experience with Secure Multi-Processing in Java.* Technical Report 560-97, Department of Computer Science, Princeton University, September, 1997.

2. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System.* 15th ACM Symposium on Operating Systems Principles, p.267-284, Copper Mountain, CO, December 1995.

3. B. Bershad, T. Anderson, E. Lazowska, and H. Levy. *Lightweight Remote Procedure Call.* 12th ACM Symposium on Operating Systems Principles, p. 102-113, Lichtfield Park, AZ, December 1989.

4. R. S. Boyer, and Y. Yu. *Automated proofs of object code for a widely used microprocessor.* J. ACM 43(1), p. 166-192, January 1996.

5. J. Chase, H. Levy, E. Lazowska, and M. Baker-Harvey. *Lightweight Shared Objects in a 64-Bit Operating System.* ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1992.

6. S. Drossopoulou, S. Eisenbach. *Java is Type Safe - Probably.* 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 1997.

7. Electric Communities. *The E White Paper.* http://www.communities.com/products/tools/e.

8. R. Engler, M. Kaashoek, and J. James O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management.* 15th ACM Symposium on Operating Systems Principles, p. 251–266, Copper Mountain, CO, December 1995.

9. General Magic. Odyssey. http://www.genmagic.com/agents.

10. L. Gong. *Java Security: Present and Near Future.* IEEE Micro, 17(3), p. 14–19, May/June 1997.

11. Gong, L. and Schemers, R. *Implementing Protection Domains in the Java Development Kit 1.2.* Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.

12. J. Gosling, B. Joy, and G. Steele. *The Java language specification.* Addison-Wesley, 1996.

13. D. Hagimont, and L. Ismail. *A Protection Scheme for Mobile Agents on Java.* 3rd Annual ACM/IEEE Int'l Conference on Mobile Computing and Networking, Budapest, Hungary, September 26–30, 1997.

14. H. Härtig, et. al. *The Performance of μ-Kernel-Based Systems.* 16th ACM Symposium on Operating Systems Principles, p. 66–77, Saint-Malo, France, October 1997.

15. G. Heiser, et. al. *Implementation and Performance of the Mungi Single-Address-Space Operating System.* Technical Report UNSW-CSE-TR-9704, Univeristy of New South Wales, Sydney, Australia, June 1997.

16. JavaSoft. *JavaBeans, Version 1.01 Specification.* http://java.sun.com.

17. JavaSoft. *Remote Method Invocation Specification.* http://java.sun.com.

18. JavaSoft. *New Security Model for JDK1.2.* http://java.sun.com

19. JavaSoft. *Java Servlet API.* http://java.sun.com.

20. JavaSoft. *JavaServer Documentation.* http://java.sun.com

21. A. K. Jones and W. A. Wulf. *Towards the Design of Secure Systems.* Software Practice and Experience, Volume 5, Number 4, p. 321–336, 1975.

22. X. Leroy. *Objective Caml.* http://pauillac.inria.fr/ocaml/.

23. H. M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Massachusetts, 1984.

24. J. Liedtke. *On μ-kernel Construction.* 15th ACM Symposium on Operating Systems Principles, p. 237–250, Copper Mountain, CO, December 1995.

25. J. Liedtke, et. al. *Achieved IPC Performance.* 6th Workshop on Hot Topics in Operating Systems, Chatham, MA, May.

26. T. Lindholm, and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

27. Microsoft Corporation and Digital Equipment Corporaton. *The Component Object Model Specification.* Redmond, WA, July 1996.

28. Microsoft Corporation. *Microsoft Security Management Architecture White Paper.* http://www.microsoft.com/ie/security.

29. G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language.* 25th ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1998.

30. G. Necula and P. Lee. *Safe Kernel Extensions Without Run-Time Checking.* 2nd USENIX Symposium on Operating Systems Design and Implementation, p. 229–243, Seattle, WA, October 1996.

31. G. Necula. *Proof-carrying code.* 24th ACM Symposium on Principles of Programming Languages, p. 106–119, Paris, 1997.

32. G. Nelson, ed. *System Programming in Modula-3.* Prentice Hall, 1991.

33. Netscape Corporation. *Java Capabilities API.* http://www.netscape.com.

34. Rashid, R. *Threads of a New System.* Unix Review, p. 37–49, August 1986.

35. D. D. Redell. *Naming and Protection in Extendible Operating Systems.* Technical Report 140, Project MAC, MIT 1974.

36. V. Saraswat. *Java is not type-safe.* at http://www.research.att.com/~vj/bug.html.

37. Z. Shao. *Typed Common Intermediate Format.* 1997 USENIX Conference on Domain-Specific Languages, Santa Barbara, California, October 1997.

38. J. S. Shapiro, D. J. Farber, and J. M. Smith. *The Measured Performance of a Fast Local IPC.* 5th Int'l Workshop on Object-Orientation in Operating Systems, Seattle, WA. 1996

39. E. G. Sirer, M. Fiuczynski, and P. Pardyak. *Writing an Operating System with Modula-3.* First Workshop on Compiler Support for System Software, Tucson, AZ, February 1996.

40. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation.* 14th ACM Symposium on Operating Systems Principles, p. 203–216, Asheville, NC, December 1993.

41. D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.

42. W. A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, NY, 1981.

The Safe-Tcl Security Model

Jacob Y. Levy
Laurent Demailly
Sun Microsystems Laboratories
jyl or demailly@eng.sun.com
John K. Ousterhout
Brent B. Welch
Scriptics Inc.
bwelch or ouster@scriptics.com

Abstract

Safe-Tcl is a mechanism for controlling the execution of programs written in the Tcl scripting language. It allows untrusted scripts (applets) to be executed while preventing damage to the environment or leakage of private information. Safe-Tcl uses a padded cell approach: each applet is isolated in a safe interpreter where it cannot interact directly with the rest of the application. The execution environment of an applet is controlled by a trusted script running in a master interpreter. Safe-Tcl supports applets using multiple security policies within an application. These policies determine what an applet can do, based on the degree to which the applet is trusted. Safe-Tcl separates security management into well-defined phases that are geared towards the party responsible for each aspect of security.

1 Introduction

Security issues arise whenever one person invokes a program written by another person. A program usually executes with all the privileges of the user who invoked it, so the program can read and write the user's files, send electronic mail on behalf of the user, open network connections, and run other programs. If a program is malicious, it can harm the user in many ways, such as by modifying the user's files, leaking sensitive information, or crashing the user's computer.

The traditional "solution" to the security problem has been for people to avoid programs written by people they don't trust. Unfortunately, two trends are making this approach less and less practical. The first trend is an increase in information sharing between people, for example via the World Wide Web; in many cases, the creator of the information is unknown to the recipient of the information. The second trend is a blurring of the distinction between programs and data, so that the act of retrieving and viewing information can cause a program associated with the data to be executed. For example, many systems allow a CD ROM disk to contain a start-up program that is run silently whenever the disk is inserted into a drive. Another example is the Java™ *language [1], which, when used in conjunction with web browsers, allows programs to be downloaded and executed locally. When a page containing a Java applet is viewed, the applet is executed locally to provide functionality that is not implemented by the browser itself. As a result of these trends, it is becoming more difficult for users to tell when they are running a program or who wrote the program.

Safe-Tcl makes it safe for people to run applets written in the Tcl scripting language [7][10] without necessarily knowing their origin or trustworthiness. Safe-Tcl avoids potential security problems by restricting the behavior of applets so that they have fewer capabilities than the users who invoke them. The privileges granted to an applet can be adjusted to match the applet's trustworthiness. Applets of unknown origin should not be trusted at

*. Sun, Sun Microsystems, Java, and the Sun logo are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries.

all, so they run with very few privileges. If the author of an applet can be authenticated, and if that author is partially or fully trusted, the applet can execute with greater privileges. The mechanisms for authentication and granting of privilege are automated, so applications such as Web browsers can use Safe-Tcl without involving the user.

In Safe-Tcl, untrusted applets are executed in separate environments that are isolated from the application. The features available to an applet are selected by the trusted portions of the application. The implementation of Safe-Tcl is based on two basic facilities: *safe interpreters*, which provide restricted virtual machines for executing applets, and *aliases*, which are used by applets to request services from the trusted portions of the application in a controlled fashion. The alias mechanism makes it possible to provide restricted access to features that are essentially unsafe, such as file or socket access.

Safe interpreters and aliases function much like the kernel space/user space mechanism that has been used for protection in operating systems for several decades. Safe interpreters correspond to the address spaces for user-level programs, and aliases correspond to kernel calls.

The Safe-Tcl security model has three particular strengths:

- Safe-Tcl separates untrusted code from trusted code, with clear and simple boundaries between environments having different security properties.
- Safe-Tcl does not prescribe any particular security policy and supports varying levels of trust. Instead, it provides a mechanism for implementing a variety of security policies and levels of trust. Organizations can implement different policies based on their needs, and a single application can use different security policies for different applets.
- Safe-Tcl gains power and flexibility by using Tcl throughout as the scripting language. All configuration information is expressed as Tcl scripts, and the mechanisms for verifying trust, checking permissions and implementing policies are also expressed in Tcl.

The rest of this paper is organized as follows. Section 2 provides background information on the Tcl scripting language. Section 3 introduces the security issues associated with executing applets. Section 4 describes the basic mechanisms of the Safe-Tcl security model, including safe interpreters, aliases, and protected commands. Section 5 discusses security policies and security packages. Section 6 explains how the master interpreter decides whether to let an applet use a given policy. Section 7 describes the overall Safe-Tcl security model. Section 8 discusses how Safe-Tcl deals with denial-of-service and privacy attacks. Section 9 gives the implementation status of Safe-Tcl, and Section 10 compares Safe-Tcl with other security models.

2 Overview of Tcl

Tcl is an interpreted scripting language [7][10]. Its simple syntax is based on *commands* made up of *words*, much like Unix shell programs such as `sh`. For example, the command

```
set a 45
```

contains three words. The first word of each command, such as `set` in the example, selects a C *command procedure* that will carry out the command, and the other words are passed to the command procedure as arguments. The Tcl language syntax consists only of a few simple substitution and quoting rules used to parse commands. Most of the behavior of Tcl is defined by the command procedures, which are free to interpret their arguments however they like.

Tcl is *embeddable* and *extensible*. The Tcl interpreter is a C library package that can be incorporated in a variety of applications. Several dozen basic commands are implemented in C as part of the Tcl interpreter. Each application can define additional Tcl commands in C,C++, or Java to augment the basic facilities provided by Tcl. Typically, an application will implement just a few Tcl commands that provide primitive access to its facilities; more complex features are created by writing Tcl scripts that combine the application's primitive features with the built-in commands.

It is also possible to create packages containing useful sets of Tcl commands implemented in C, C++, or Java and then load these packages into any Tcl application on the fly. Tk is one such extension; it provides a collection of commands for creating graphical user interfaces.

Tcl has four properties that make it attractive as a

vehicle for executing untrusted scripts:

- The language is interpreted. All actions are already mediated, so this is a natural place to add security controls.
- The language is safe with respect to memory references: it has no pointers, array references are bounds-checked, and storage is managed automatically by the Tcl interpreter.
- Interpreters are first-class objects. An interpreter consists of a set of Tcl commands, a set of variable values, and an execution stack. An application can contain several interpreters that are disjoint from each other. This makes it possible to isolate scripts with different security properties in different interpreters.
- The language is command-oriented; the facilities available to a Tcl script are determined by the set of commands defined in its interpreter. Access to unsafe features is controlled by curtailing access to specific commands in an interpreter executing an untrusted script.

Our work addresses security issues in Tcl scripts and assumes that the implementation of the Tcl interpreter is trustworthy. Extensions written in C, C++, or Java are not available to untrusted scripts unless the extension writer provides a special initialization procedure that restricts access to unsafe commands in the extension.

3 Security Issues

The security issues associated with applets fall into four major groups: integrity attacks, privacy attacks, impersonation attacks and denial of service attacks:

- Integrity Attacks: a malicious applet may try to modify or delete information in the environment in unauthorized ways. For example, it might attempt to transfer funds from one account to another in a bank, or it might attempt to delete files on a user's personal machine. In order to prevent this kind of attack, applets must be denied almost all operations that modify the state of the host environment. Occasionally, it may be desirable to permit the applet to make modifications; for example, if the applet is an editor, it might be allowed to write to a file if approved by a user through a file selection dialog.
- Privacy Attacks: these attacks try to steal or leak information belonging to the user. A malicious applet may try to read private information from the host environment and transmit it to a conspirator outside the environment. Information disclosed in this way may have direct value to the recipient, such as business information that could affect the price of a company's stock, or its disclosure could damage the party from which it was taken, for example, if it describes an individual's treatment for substance abuse.
- Impersonation Attacks: a malicious applet might perform actions on behalf of the user of the hosting application without his or her authorization. For example, it may send e-mail in the user's name containing damaging statements, or it may make it appear that the user is attempting to mount some form of attack against a remote resource. The purpose of an impersonation attack can be to damage the impersonated user's reputation.
- Denial of Service Attacks: these attacks interfere with the normal operation of the host system. For example, an applet might consume all the available file space, cover the screen with windows so that the user cannot interact with any other applications, or exercise a bug to crash its hosting application.

It is unlikely that any security policy can completely eliminate all security threats. For example, any bug in an application gives a malicious applet the opportunity to deny service by crashing the application. In addition, there exist subtle techniques for signaling that make it nearly impossible to protect the privacy of information once it has been given to an applet [4]. Attempts to completely eliminate the risks would restrict applets to such a degree that they would not be able to perform any useful functions.

Thus, Safe-Tcl does not try to eliminate security risks entirely. Instead, it attempts to reduce the risks to a manageable level, so that the benefits provided by applets are greater than the costs incurred by security attacks. Safe-Tcl concentrates on preventing integrity attacks, privacy attacks and impersonation attacks. It is not geared towards preventing all denial of service attacks. Denial of service attacks generally do not permanently impact the user's ability to perform useful work or the integrity of her information, and thus, while annoying, are less damaging.

4 Safe Tcl

Safe-Tcl uses a *padded cell* approach to security:

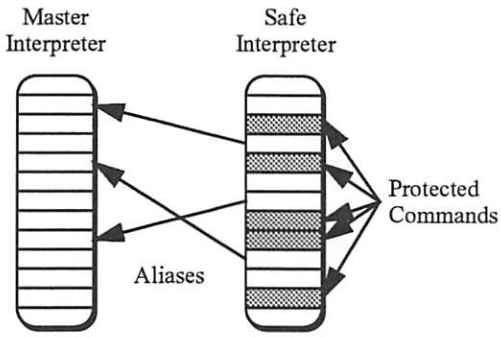


Figure 1. The basic Safe-Tcl mechanisms. Trusted scripts execute in the master interpreter while untrusted applets execute in the safe interpreter. All unsafe commands in the safe interpreter are protected so that they cannot be invoked from the safe interpreter. Aliases provide a mechanism for the applet to request mediated operations from the master. The master interpreter can invoke the protected commands in the safe interpreter.

applets are executed in isolated environments where their capabilities can be restricted. Padded cells are implemented using three mechanisms that are shown in Figure 1. First, Safe-Tcl uses *safe interpreters* to isolate applets and prevent them from using any of the unsafe features of the language. Then it restores access to a restricted subset of the unsafe features using *aliases* and *protected commands*.

If a Tcl application wishes to execute an applet, it uses two interpreters: a master interpreter and a safe interpreter. The master interpreter retains full functionality, so only trusted scripts such as those written by the user or the application designer may execute there. The safe interpreter is used for executing the applet. All of the unsafe commands (those that could result in security compromises if misused) are disabled in the safe interpreter. These commands include those for accessing the file system, executing subprocesses, opening sockets, and many more. A script that tries to use the disabled features will get runtime errors.

The set of commands left in the safe interpreter, called the *safe base*, allows the applet to perform only safe actions. With only this set of commands an interpreter is indeed safe for executing applets, but in this state the interpreter is not very interesting because scripts running in it are completely isolated. If a script cannot access files, open sockets, or communicate with other processes, then there aren't many useful things that it can do. In fact, most useful programs involve activities that are unsafe in the general case. In order for applets to perform useful activities, they must have restricted access to unsafe functions. For example, it is not safe to let an applet write arbitrary files, but it probably is safe to let an applet create a single new file of limited size containing the results of its computation.

This separation of trusted code from untrusted code is similar to the separation in operating systems between user-space code and kernel-space code. Kernel-space code can write directly on every location on the disk, but user-space code has no direct access to the disk. Instead, it must use a system call to write data to portions of the disk as permitted by checks enforced by code executing in kernel-space.

The alias mechanism in Safe-Tcl is analogous to system calls in operating systems. An alias is an association between a command in the safe interpreter, called the *source command* for the alias, and a command in the master interpreter, called the *target*. Whenever the source command is invoked by a script in the safe interpreter, the target command is invoked instead. The target command is typically a Tcl procedure. It receives all of the arguments from the source command and its result is returned to the safe interpreter as the result of the source command.

The master interpreter has complete control over the safe interpreter. It can initiate the execution of scripts, create and delete aliases, and control the names of the source and target commands for each alias. The safe interpreter cannot create new aliases on its own. During the execution of an alias, the master can access the state of the safe interpreter and invoke additional scripts in the safe interpreter

```
# Create an array in which the names of elements are host
# names and the values are lists of acceptable port numbers.

set safeSockets(sage.eng) 1024
set safeSockets(sunlabs.eng) 80
set safeSockets(www.sun.com) {80 8015}
set safeSockets(bisque.eng) {3000 4000 5000}

# Create an alias that causes the AliasSocket command to be
# invoked in the master whenever socket is invoked in the safe
# interpreter.

interp alias $safe socket {} AliasSocket $safe

# Define the procedure that implements the alias.

proc AliasSocket {safe host port} {
    global safeSockets
    if {![info exists safeSockets($host)]} {
        error "Unknown host: $host"
    }
    if {[lsearch -exact $safeSockets($host) $port] < 0} {
        error "Bad port: $port"
    }
    return [interp invokehidden $safe socket $host $port]
}
```

Figure 2. When this code is executed in a master interpreter, it creates an alias that allows a safe interpreter to open sockets to a restricted set of addresses. Whenever the `socket` command is invoked in interpreter `$safe` the `AliasSocket` command will be invoked in the master interpreter with the name of the safe interpreter as its first argument. Thus, if the value of `$safe` is `child`, and the command "`socket bisque.eng 4000`" is invoked in the safe interpreter, then the command "`AliasSocket child bisque.eng 4000`" will be invoked in the master. The `AliasSocket` procedure checks to see if the host and port are among those that are allowed. If so, it invokes the hidden `socket` command in the safe interpreter to actually open the network connection.

to carry out the functions of the alias.

The commands that are disabled in the safe base are not actually removed from the safe interpreter; they are *protected* so that they can be invoked only by the master interpreter. This allows the master interpreter to ensure that only a subset of the command's features are used. To use a protected command, an applet must use an alias which checks that the arguments and intended usage are safe, and then the alias invokes the protected command. Figure 2 shows an alias that allows sockets to be opened only to a pre-specified list of hosts and ports. The `socket` command, which is used to create network connections, is unsafe so it is protected in the safe interpreter; the code in the figure creates a new `socket` command that is an alias. The alias validates the host and port, then invokes the protected `socket` command in the safe interpreter. To the applet the `socket` command appears to work in the normal fashion except that only certain network addresses may be used. Note that two versions of `socket` exist in the safe interpreter: the protected command and the alias.

Protected commands are needed because many Tcl commands implicitly modify the interpreter in which they are invoked. For example, the `socket` command creates a new I/O channel for use in communicating over the socket. The channel is created in the interpreter where the `socket` command executes, so if the alias invoked `socket` in its own interpreter (the master) then the safe interpreter wouldn't be able to use the resulting channel.

5 Security Policies and Security Packages

In Safe-Tcl, aliases are grouped into security policies. Each security policy has a name and contains one or more aliases that are known to be safe for certain kinds of applets. An applet chooses which policy it uses, subject to the checks described in Section 6; a single applet may only use one policy over its lifetime. Many different policies are possible, each imposing a different set of restrictions on applets controlled by the policy. Some policies are safe for all applets to use, while other policies are only safe for applets that can be verified to originate from a trusted source. We designed Safe-Tcl to encourage the development of many different policies, and to allow the reuse of policies in many applications

Why is it important to allow multiple security policies? Wouldn't it be better to have just one policy that includes all of the features that are safe for applets? Multiple security policies are needed because safe features do not compose: if feature A is safe and feature B is safe, the combination of A and B is not necessarily safe. For example, it is safe for an applet to open network connections outside the firewall as long as the applet cannot communicate with hosts inside the firewall. It is also safe for an applet to read local files, as long as this is the only communication the applet makes outside its interpreter. However, an applet that has access to both of these features can transmit local files outside the firewall, which is a breach of privacy.

Since safe features do not compose, no single security policy can include all of the features that are safe in isolation. Safe-Tcl encourages the development of many security policies, each tailored to support a different class of applets. The simplest security policy consists of just the safe base with no additional features enabled. Most security policies will probably enable a small set of additional features. In an extreme case where the applet is completely trusted, it can be given a security policy that restores the full set of unsafe Tcl commands and enables all the features provided by the hosting application.

An applet obtains a security policy by invoking the `policy` alias which is installed as part of the safe base. The alias checks whether the applet is allowed to use the requested policy. If allowed, it installs the aliases specified by the requested policy and records information that will be used later to control how these aliases are used by the applet. An applet may obtain at most a single security policy over its lifetime; once it has successfully obtained one policy it may not obtain any other policy. Changing the security policy for an applet or allowing it to use multiple policies composes the features of the security policies, which is not safe.

We expect that many policies will be similar in the set of features they provide. To encourage reuse of the implementation of aliases, Safe-Tcl has the concept of *security packages*, named sets of aliases that are installed as a unit. The aliases are implemented by one Tcl script and reused by name in multiple policies.

Policies are written in Tcl using a style that is easily parsed by the configuration management package provided with Safe-Tcl. Each policy is organized into a number of sections, and each section contains permissions and restrictions referring to a set of features. Below is a snippet of the home policy which allows an applet to communicate with servers on the host from which it was loaded, and to fetch web pages from that host:

```
section features
allow url
allow network
allow persist

section urls
allow $originHomeDirURL*
```

The *home* policy enables the *persist*, *network* and *url* security packages in the *features* section. The *persist* security package allows an applet to store a limited amount of information persistently on the user's machine, *network* allows the applet to open sockets to a restricted set of remote servers, and *url* allows access to a limited set of web pages and remote web services. The *urls* section allows URLs to be fetched from the subtree of the web site rooted at the directory containing the applet.

The *outside* policy is similar to the *home* policy and reuses the persist, network and url security packages. Its *url* section allows URLs to be fetched from the C|Net web site:

```
section features
allow persist
allow url
```

```
allow network

section urls
allow http://www.cnet.com/*
```

Aliases installed into the safe interpreter housing the applet allow the applet mediated access to features provided by the hosting application. The target command checks the arguments according to the restrictions imposed by the applet's policy and decides whether to allow the call. In the above example, if a URL fetch is requested, it is only allowed if it refers to a web page on the C|Net web site. If the operation is allowed, the call is forwarded to the actual implementation.

The Tcl web browser plug-in comes with several policies, including *home* and *outside*, mentioned above. Each of these policies is fairly restrictive, yet supports a large class of interesting applets. An advantage of having more restrictive security policies is simplicity. If a security policy included a large number of features, it would be difficult to analyze all of the interactions between its features to uncover security loopholes. A policy that includes only a small number of features is more amenable to analysis and increases our confidence that it is really secure.

6 Controlling the Use of Security Policies

Safe-Tcl is designed to support a large range of security policies. Some policies can be used by all applets, without requiring that the applet be trusted. Other policies, especially those that provide access to features that can be used to mount security attacks, require that the applet be trusted to some degree before they can be used by that applet. Safe-Tcl checks whether an applet is allowed to use a policy when the applet first requests to use the policy. If the applet is allowed to use the policy, the security packages enabled by the requested policy are installed.

Safe-Tcl provides the concept of a *trust map* to allow site and application administrators to control which applets can use each security policy. A trust map is a Tcl script, organized into sections similarly to a policy, that specifies under what circumstances each policy can be used; the map also defines the names of all security packages provided by the application and controls various other application level resources. Each application using Safe-Tcl has its own trust map; here is part of the policies section in the trust map for the Tcl web browser plug-in:

```
section policies
disallow trusted
allow home
allow javascript \
ifallowed javascriptTrustedURLs
  $originURL
```

A trust map can contain statements that disable a policy for all applets, as is the case for the *trusted* policy, above. Similarly, a policy can be enabled for all applets, as for the *home* policy. The trust map also provides a place to insert authenticators that decide to allow or disallow the use of a policy based on some property of the applet. We see an example of this in the statement allowing the use of the *javascript* policy if the URL from which the applet was loaded is allowed by the section *javascriptTrustedURLs*. Authenticators can use attributes of the applet such as its MD5 checksum [9], an attached signed certificate [5], or the URL from which it was loaded.

This separation of trust and authentication from the actual policies is important, because otherwise policies can not be shared between applications.

7 Security Model and Security Roles

Safe-Tcl cleanly factors into three distinct parts that reflect the roles of three human participants in the management of security:

- Security packages encapsulate features provided by applications and implement constrained access to these features. Security packages are provided by the application's author.
- Security policies determine which security packages an applet can use and what resources it can access using the provided features. A security expert designs each security policy.
- The trust map determines whether an applet can use a given security policy. Trust maps are edited by site and application administrators.

Much of the power and flexibility of Safe-Tcl stems from its use of Tcl throughout to implement the security model. Tcl is used to implement the configuration mechanism that controls access to resources

by applets. Tcl scripts specify whether a policy can be used by an applet and under what conditions. Finally, Tcl is used to implement the security packages and the aliases used by applets to access features provided by security packages.

8 Denial-of-Service and Privacy Attacks

Although Safe-Tcl was designed primarily to address issues of integrity, impersonation and privacy, its mechanisms can also be used to prevent denial-of-service attacks. For example, an applet can be prevented from consuming all the disk space by protecting the `puts` command, which writes data to files. In its place an alias can be created to count the bytes that are output and enforce a limit.

However, many denial-of-service attacks, particularly those associated with graphical user interfaces, are hard to prevent. For example, an applet could attempt to create a window that covers the whole screen and prevent the user from interacting with any other applications. Aliases and hidden commands could be used to restrict the sizes of windows, but the applet could then create several smaller windows that together cover the whole area of the screen. Furthermore, in some situations (such as laptop computers with small screens) it may be desirable to let an applet use the entire screen.

Safe-Tcl currently does not prevent most denial-of-service attacks. We will address this in the future with a combination of resource controls and a kill-key that lets a user intervene when an applet misbehaves.

Lampson [4] shows that it is generally impossible to prevent information from being communicated from one applet to another through covert channels such as manipulation of the scheduler or other finite accessible resources. While the rate of communication is limited, it is still possible to leak a significant number of bits per second through this form of attack. For example, if one applet has access to a file stored on a server within a firewall, while another applet has the ability to communicate over the network with a server outside the firewall, it is possible for the two applets to collaborate and disclose information stored in these files to outside parties. Safe-Tcl by default disallows untrusted applets to use resources on an Intranet. Thus, largely, information leakage through covert channels is prevented because applets by default cannot gain access to private information stored on hosts on an Intranet.

9 Status

Safe-Tcl has been available in public Tcl releases since the Tcl 7.5 release in April 1996. Safe-Tcl integration with Tk is implemented as part of a Tcl/Tk plug-in module for web browsers which was released in July 1996 [6]. The plug-in allows Tcl/Tk scripts to be included in Web pages with embedded custom GUIs. The 2.0 plug-in release made in January 1998 offers full support for safe interpreters, aliases, mechanisms for creating and installing security policies, and a trust map implementation. Safe-Tcl does not yet support a kill key, CPU usage limits, or authentication.

9.1 Performance

Table 1 shows a few measurements of the performance of Safe-Tcl. Overall, Safe-Tcl does not add substantially to the execution time of an application. Our experience with the performance of the Safe Tcl security model in Tcl 8.0 is that it does not add noticeable overhead. Table 1 shows a few measurements that indicate the overhead of invoking an alias is about twice that of calling a null Tcl procedure. The difference between calling a null alias and and alias to **pid**, which returns a value, shows the cost of marshalling parameters and results between interpreters. For each benchmark we measured calling it directly versus calling the same code via an alias from another interpreter. The benchmarks are calling the built in **pid** procedire. a null procedure, a procedure taking ten arguments, a procedure that adds its ten numeric arguments and a ten element list reversal procedure.Measurements were taken on a Pentium 233 running Linux.

Table 1:

Command	usec
call pid	5
call alias to pid	10
null proc	7
call alias to null proc	11

Table 1:

Command	usec
10arg procl	14
alias to 10arg proc	19
10add proc	26
alias to 10add proc	31
lreverse proc	213
alias to lreverse proc	228

10 Related Work

10.1 The Borenstein/Rose prototype

Nathaniel Borenstein and Marshall Rose implemented a prototype of Safe-Tcl in 1992 that pioneered most of the ideas, including safe interpreters and aliases [2]. The Borenstein/Rose prototype was used for active e-mail messages and later as part of the First Virtual Holdings Internet payment system.

Our implementation generalizes the Borenstein/Rose prototype in several ways. The prototype only allowed one safe interpreter, while our work allows any number of safe interpreters. There was no concept of security policies and security packages in the Borenstein/Rose prototype, and there were no mechanisms to specify configuration information. Their implementation was specific for one problem domain, how to send scripts via electronic mail messages safely, while our approach can be applied to any problem domain. Finally, our work introduces the concept of trust maps to allow or disallow applets to use specific policies.

10.2 Object Oriented Systems.

Most other security models for executing untrusted code, such as Java [12][13] and Telescript [11], are based on object systems. These models are similar to Safe-Tcl in that they use safe languages that control pointers and memory allocation. However, they differ from Safe-Tcl in that they provide only a single virtual machine that contains all of the objects and classes. Security properties are associated with individual objects or classes; for example, one class may be marked as coming from an untrusted source while another may be marked as trusted. This infor-

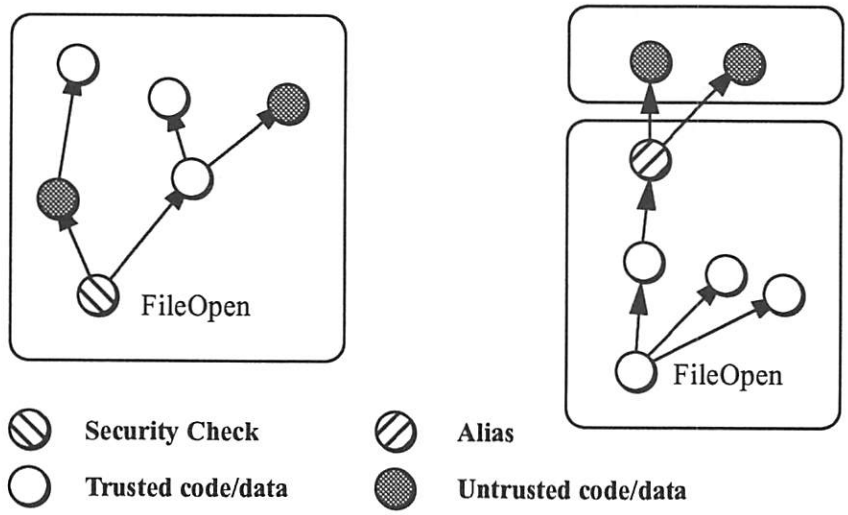


Figure 1. With an object-oriented approach to security (left) security checks must be done at a low level, guarding the call to the operating system, so they cannot be by-passed. A low-level check makes it difficult to determine if trusted code called by untrusted code should be allowed to perform the operation. With the padded cell approach (left) security checks can be done at a high-level when control transfers from code in a separate virtual machine. Edges in the graph represent method or procedure calls, and boxes represent virtual machine boundaries.

mation is used when deciding whether or not to allow a particular operation. For example, before allowing a file to be opened, Java checks to see if there are any untrusted classes on the current call stack; if so, the open operation is denied. In contrast, Safe-Tcl's padded cell approach uses multiple virtual machines (interpreters) and the security properties are associated with the virtual machine, not individual pieces of data or code. Security decisions are made based on the virtual machine that is currently executing; thus, while executing in a safe interpreter it is not possible to open a file, but it is possible to open a file if control is first transferred to a trusted interpreter using an alias.

In Java the use of a single virtual machine for both trusted and untrusted code requires security checks at a low level within the system, and this make it difficult to implement sophisticated security policies. Security manager calls must be made at low levels, right before a native method call into the operating system, otherwise untrusted code could call the method directly and bypass the security check. The security check can easily test if untrusted code is on the call stack and deny access. However, it is more difficult to implement a policy that denies some accesses by untrusted code but allows other accesses. For example, a policy could provide a method that displays a file selection dialog and returns an open I/O channel. The goal is that the file open operation is only done via the user interface dialog so the user knows what file is being accessed. With one low-level check, it would be difficult to allow accesses from the trusted dialog without allowing other, more direct accesses from the untrusted code. The use of a single virtual machine makes it difficult to hide dangerous operations, which forces a low-level security check. The low-level check limits the flexibility of policy code and adds overhead to dangerous operations for both trusted and untrusted code.

In contrast, in Safe-Tcl security checks are done at a very high-level, when an alias transfers control to a trusted virtual machine, which leaves room for flexible policy implementations. (See Figure 1.) Safe-Tcl can replace the open command with an alias that displays a user interface dialog or uses policy code that limits untrusted code to a private directory and a limited number of files. The implementation of these aliases may look through the file system and open various configuration files that specify the directory location and limits on files. Eventually the alias will open a file for use by the untrusted code. The alias is implemented in a trusted virtual machine that can do anything on behalf of untrusted code. This provides a very flexible environment for wrapping policy code around dangerous operations, and the policies only add overhead to untrusted code that must use aliases.

There are more advantages to the Safe-Tcl implementation:

- Aggressive security policies can hide more commands from untrusted virtual machines, such as those that gave clock and timing information. This only affects the untrusted virtual machine, and it does not require modification of the existing clock and timer implementations. In Java, adding a security manager check to the clock subsystem requires modification of trusted code.
- Tcl allows multiple virtual machines (i.e., interpreters) within the same application, and different interpreters can have different security policies through different sets of aliases. This is more awkward in Java because there is a single security manager that would have to manage different policies for different kinds of untrusted classes. Our understanding of Java development is that they plan to support multiple security managers in the future.

10.3 Pure authentication: ActiveX

An approach proposed by Microsoft authenticates downloaded applets and asks the user of the hosting machine to assign trust to the identified principal. ActiveX prevents untrusted programs from being executed. In this approach all security decisions are delegated to the user of the machine. This is the only approach that works for compiled programs, because there is no practical mechanism for restricting what machine code can do.

But ActiveX only verifies identity, and trust involves more than just authentication. Authentication identifies the principal (person or organization) who wrote something, but it doesn't indicate whether the principal is trustworthy. Trust can really only be placed in principals you are *familiar* with. The authentication approach works well for applets written by large companies that are known to be trustworthy (or that can be sued if their software is defective). However, authentication doesn't

help when applets are written by individuals and smaller companies that are not well known. One of the reasons for the popularity of the World Wide Web is that it enables communication among large numbers of individuals and small organizations that have no prior knowledge of each other.

ActiveX is unsuitable for executing untrusted programs retrieved from the web, because it is based only on authentication and does not provide any restrictions on what a program can do once it is trusted. This does not scale well to the web's distributed and decentralized nature. In contrast, Safe-Tcl enables safe execution of code trusted to varying degrees, ranging from completely untrusted to completely trusted. For some operations, no knowledge about an applet author's identity is needed, while other operations may require full authentication.

11 Conclusions

There is no silver bullet that will make security trivial. Creating safe environments for executing applets will always be difficult, and no security model will ever be totally safe, since even a small bug in programming can open a huge security hole. However, we think it is possible to create environments where applets with varying degrees of trust can be executed with an acceptable level of risk. Safe-Tcl has several properties that simplify the creation of such environments:

- The padded cell model is simple. It generalizes the user space-kernel space model that has been used successfully in operating systems for several decades.
- Safe-Tcl groups data and code with similar security properties together, which reduces the amount of code that must be aware of security issues.
- Safe-Tcl separates security management into well-defined phases that are geared towards the party that is responsible for each aspect of security. It separates implementation of security policies, generally an activity for security experts, from the implementation of security packages, which is done by engineers creating an application, and from configuration management, an activity reserved for site and system administrators.
- Authentication is important for secure systems but it is not sufficient by itself to provide protection. The mechanisms of Safe-Tcl provide a well defined way to constrain the capabilities of code after its origin has been authenticated.

Our experiences with Safe-Tcl have taught us three important lessons about security. The first is that safe features do not necessarily compose. This makes it difficult to provide a single security policy with a large variety of features; instead, it encourages a large number of smaller, specialized security policies. The second lesson is that it is important to take advantage of authentication mechanisms yet not require them. If programs are to be intimately tied to information, and if information is to be freely distributed among strangers, then it is important to support the execution of totally untrusted programs. At the same time, authentication can be used to boost the power of applets when they come from known sources. The third lesson is that using a scripting language to implement a security model is both doable and adds unique value by allowing the resulting system to be very flexible and configurable.

12 Acknowledgments

This work would never have come about without the pioneering efforts of Nathaniel Borenstein and Marshall Rose, who designed and built the Safe-Tcl prototype. Nathaniel Borenstein, Wan-Teh Chang, Robert Drost, Clif Flynt, Li Gong, Mark Harrison, Ray Johnson, Anand Palaniswamy, Marshall Rose, Rich Salz, Juergen Schoenwaelder, and Glenn Vanderburg provided useful comments that improved the presentation of this paper.

13 References

[1] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, ISBN 0-201-63455-4, 1996.

[2] N. Borenstein, "E-mail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail," *IFIP WG 6.5 Conference*, Barcelona, May, 1994, North Holland, Amsterdam, 1994.

[3] D. Denning and P. Denning, "Data Security," *Computing Surveys*, Vol. 11, No. 3, September 1979, pp. 227-249.

[4] B. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, Vol.

16, No. 10, October 1973, pp. 613-615.

[5] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM Transactions on Computer Systems*, Vol. 10, No. 4, November 1992, pp. 265-310.

[6] J. Levy, *Welcome to the Tcl Plug-in*, `http://sunscript.sun.com/plugin/`.

[7] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-63337-X, 1994.

[8] R. Rivest, *The MD5 Message Digest Algorithm*, RFC 1321, April 1992.

[9] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," Proc. 14th Symposium on Operating Systems Principles, *Operating Systems Review*, Vol. 27, No. 5, December, 1993, pp. 203-216.

[10] B. Welch, *Practical Programming in Tcl and Tk*, Prentice-Hall, ISBN 0-13-616830-2, Second edition, 1997.

[11] J. White, *Telescript Technology: The Foundation for the Electronic Marketplace*, white paper, General Magic, Inc., 1994.

[12] F. Yellin, "Low Level Security in Java," *World-Wide Web Conference*, Boston MA, December 1995. Also available as http://www.javasoft.com/sfaq/verifier.html.

[13] Li Gong et al., "Going Beyond the Sandbox: an Overview of the New Security Architecture in the Java Development Kit 1.2", USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8-11, 1997.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Javan and C++, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as ten technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- 20% discount on all titles from O'Reilly & Associates.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Sage Science Press, and John Wiley & Sons.
- Special subscription rate for *The Linux Journal* and *The Perl Journal.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Advanced Resources
ANDATACO
Apunix Computer Services
Auspex Systems, Inc.
Boeing Commercial
CyberSource Corporation
Digital Equipment Corporation
Earthlink Network, Inc.
Hewlett-Packard
Internet Security Systems
Invincible Technologies
Lucent Technologies, Bell Labs
Motorola Global Software
Nimrod AS
O'Reilly & Associates
Sun Microsystems, Inc.
UUNET Technologies, Inc.
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
Digital Equipment Corporation
ESM Services, Inc.
Global Networking and Computing, Inc.
Great Circle Associates
OnLine Staffing
O'Reilly & Associates
Sprint Paranet
Texas Instruments, Inc.
TransQuest Technologies, Inc.
UNIX Guru Universe

ISBN 1-880446-94-4